Eetu Nevalainen

AJONAIKAINEN MUISTINVALVONTA YLIVUOTOJEN HAVAITSEMISEEN

AJONAIKAINEN MUISTINVALVONTA YLIVUOTOJEN HAVAITSEMISEEN

Eetu Nevalainen
Opinnäytetyö
Kevät 2018
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, langattomat laitteet

Tekijä: Eetu Nevalainen
Opinnäytetyön nimi: Ajonaikainen muistinvalvonta ylivuotojen havaitsemiseen
Työn ohjaaja: Ensio Sieppi ja Alpo Leinonen
Työn valmistumislukukausi ja -vuosi: Kevät 2018
Sivumäärä: 42 + 6 liitettä

Opinnäytetyön aiheena oli ajonaikainen muistinvalvonta muistin ylivuotojen havaitsemiseen. Tavoitteena oli kehittää menetelmä suojamuuttujien lisäämiseen lähdekoodiin puskureiden viereen muistissa, jotta mahdolliset ylivuodot havaittaisiin.

Työtä varten kehitettiin sovellus, jota käyttämällä lähdekoodin puskureiden viereen muistissa voidaan määrittää suojamuuttujia.

Suojamuuttujat ja sovelluksen toiminta testattiin Windows- sekä Linux-ympäristöissä yksinkertaisella C-ohjelmointikielellä kirjoitetulla ohjelmalla, jossa puskureihin aiheutettiin tarkoituksellisesti ylivuotoja. Sovellus onnistui luomaan suojamuuttujat määrättyihin kohtiin lähdekoodia ja jokainen aiheutettu ylivuoto havaittiin.

Asiasanat: ohjelmistokehitys, tietokannat, prosessimuisti

ABSTRACT

Oulu University of Applied Sciences
Degree programme in Information Technology, Wireless Devices

Author: Eetu Nevalainen
Title of thesis: Runtime memory monitoring for overflow detecting
Supervisor: Ensio Sieppi and Alpo Leinonen
Term and year when the thesis was submitted: Spring 2018
Pages: 42 + 6 appendices

The purpose of this thesis was to develop a method for injecting guard variables next to source code's buffers in memory so that the possible overflows could be detected.

A desktop application was made for the purpose of the guard variable injections.

The guard variables and the desktop application were tested on both Windows and Linux with a simple C program where buffer overflows were caused intentionally. The desktop application injected guard variables into the definite positions in the source code and all of the buffer overflows were detected.

Keywords: software development, database, process memory

SISÄLLYS

1 JOHDANTO

Puskurin ylivuoto on tila, jossa ohjelma kirjoittaa dataa puskurille varatun muistin ulkopuolelle. Puskurin ylivuoto voi korruptoida muistissa olevaa dataa tai aiheuttaa prosessissa epämääräisiä tiloja. (1, s. 35–36.) Suojamuuttuja on tavallinen muuttuja lähdekoodissa, jonka tehtävä on sijaita muistissa puskurin vieressä ja kaapata osa mahdollisesta ylivuodosta itseensä. Mikäli suojamuuttujalle alustettu arvo muuttuu, on suojamuuttuja kaapannut itseensä ylivuodon.

Työn vaatimuksena on kehittää menetelmä globaalien ja lokaalien puskureiden suojaamiseen suojamuuttujilla. Suojamuuttujien tilat tulee voida tarkistaa erillisestä käskystä tai kehittäjän määrittämästä lähdekoodin kohdasta. Mikäli tarkistettu suojamuuttuja on kaapannut ylivuodon, tulee kehittäjän saada tietoa kyseisestä suojamuuttujasta, jotta suojamuuttuja voidaan paikantaa lähdekoodista. Menetelmällä tullaan testaamaan kehitysvaiheessa olevia ohjelmistoja sekä jo valmiita ohjelmistoja puskureiden ylivuodoilta.

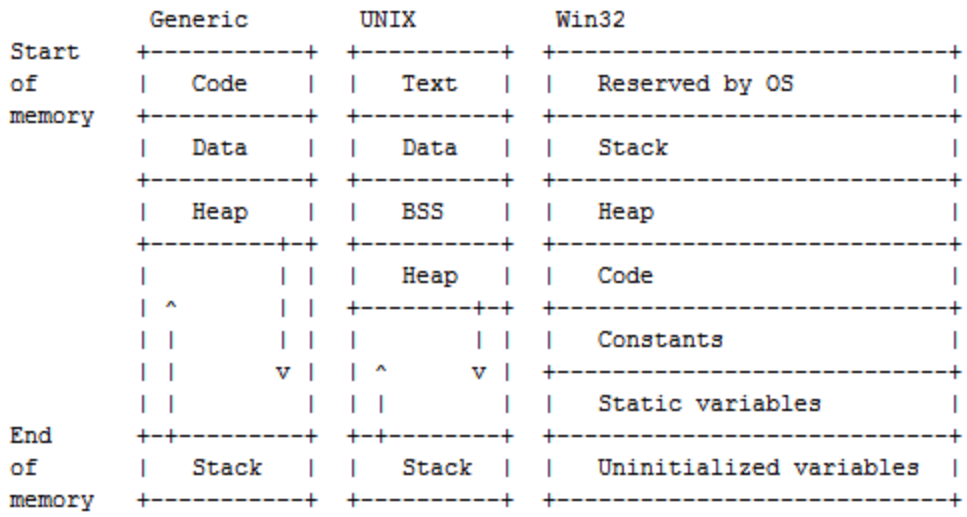
Työn toimeksiantajana toimii Tracker Oy, joka kehittää, valmistaa ja markkinoi GPS-paikantimia erityisesti metsästyskoirien paikantamiseen. Tracker on kommunikaatiojärjestelmä metsästäjille ja ulkoilmaihmisille, joka koostuu pilvipalvelusta, seurantaohjelmasta ja koirapaikannuspannasta. Tracker mahdollistaa maastossa liikkuvan koiran paikantamisen reaaliajassa sekä kommunikaation seurueen jäsenten kesken.

Työtä tehtiin Tracker Oy:n tiloissa Linux-ympäristössä ja kotona Windows-ympäristössä.

2 PUSKURIN YLIVUOTO

2.1 Prosessimuisti

Prosessi on keskusmuistiin ladattu ohjelma, jonka suoritusta hallitsee käyttöjärjestelmä. Prosessimuisti koostuu tyypillisesti koodi-, data-, keko- ja pinosegmenteistä (kuva 1). Koodisegmenttiä kutsutaan myös nimellä tekstisegmentti. Koodisegmentti sijaitsee kirjoitussuojatulla muistin alueella, jonka muokkaaminen aiheuttaa virhesignaalin. (1, s. 36.)



KUVA 1. Prosessimuisti eri järjestelmissä (1, s. 36)

Datasegmentti sisältää ohjelman vakiomuuttujat, alustetut globaalit muuttujat ja alustetut staattiset muuttujat. BSS-segmentti (block started by symbol) sisältää ohjelman alustamattomat globaalit ja staattiset muuttujat. Alustamattomat muuttujat eritellään alustetuista muuttujista, jotta alustamattomia muuttujia ei tarvitse kirjoittaa objektitiedostoon. (1, s. 78.) BSS-segmentin muistialueen tavut alustetaan nolliksi, joten ohjelmassa määritellyt alustamattomat staattiset ja globaalit muuttujat saavat alkuarvoikseen nollan (2, s. 456).

Kuva 2 havainnollistaa eri muuttujatyyppien sijoittumiset segmentteihin.

```c
static int staattinen_alustettu_globaali = 8; // datasegmentti
static int staattinen_alustamaton_globaali; // BSS-segmentti

int main()
{
    int alustamaton_lokaali; // pinosegmentti
    int alustettu_lokaali = 3; // pinosegmentti
    static int staattinen_alustamaton_lokaali; // BSS-segmentti
    static int staattinen_alustettu_lokaali = 5; // datasegmentti
    int *dynaamisesti_varattua_muistia = (int*)malloc(64); // kekosegmentti
    return 0;
}
```

KUVA 2. Muuttujien segmentit

Keko on muistialue, josta voidaan varata muistia dynaamisesti ohjelman muuttujille ajonaikana. C-ohjelmointikielessä keosta varataan muistia funktiolla malloc() ja varattu muisti vapautetaan funktiolla free(). Ohjelman muistiin latauksen yhteydessä keko jätetään yleensä alustamatta, joten keosta varattu muisti aiheuttaa muuttujalle määrittämättömän tilan. (2, s. 456.)

Pinosegmentti on struktuuriltaan dynaaminen LIFO (last-in, first-out), johon työnnetään kutsuttavan funktion parametrit ja muuttujat sekä paluuosoite. Paluuosoite osoittaa seuraavaan suoritettavaan käskyyn, jota edeltää funktion kutsu. Kun funktio päättyy, luetaan pinosta paluuosoite, josta ohjelma jatkaa suoritusta. (1, s. 37.)

2.2 Puskurin ylivuoto

Puskurin ylivuoto on tila, jossa ohjelma kirjoittaa dataa puskurille varatun muistin ulkopuolelle. C- ja C++-ohjelmointikielillä ohjelmoidut ohjelmat altistuvat puskureiden ylivuodoille tyypillisesti seuraavista syistä:

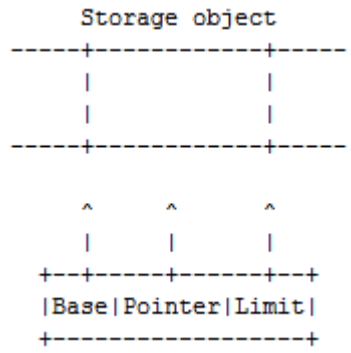
- Varatun muistin rajojen ylittämistä ei tarkisteta.
- Merkkijonot eivät pääty null-merkkiin.
- Standardi-kirjasto tarjoaa merkkijonojen käsittelyyn funktioita, jotka eivät huomioi puskurin kokoa. (1, s. 35–36.)

Riippuen ylivuodon sijainnista muistista sekä sen koosta, voi puskurin ylivuoto jäädä havaitsematta ja samalla korruptoida muistissa olevaa dataa, aiheuttaa prosessissa epämääräisiä tiloja tai päättää prosessin epänormaalisti. Puskurin ylivuodot ovat hankalia ongelmia ohjelmistokehityksessä, koska ne voivat jäädä havaitsematta ohjelman kehitys- ja testausvaiheissa. (1, s. 35–36.)

2.3 Malleja puskurin ylivuodon havaitsemiseen

2.3.1 Ajonaikainen rajojen tarkistaminen

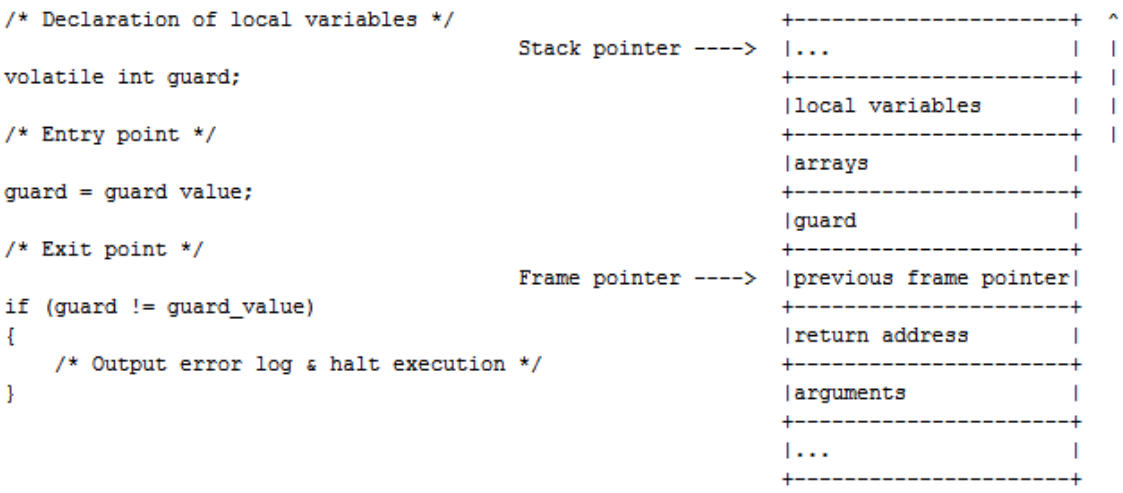
Ajonaikaisessa rajojen tarkistamisessa jokainen luotu osoitin sisältää tietoa sen kantaosoitteesta sekä viimeisestä osoitteesta, johon osoitin saa sallitusti osoittaa (kuva 3). Jotta osoittimen koko muistissa säilyy muuttumattomana, tulee osoittimen osoittaa taulukkoon, joka sisältää edellä mainitut tietueet sekä itse osoittimen. Muussa tapauksessa muistin varausta ja koodin generointia on muutettava. Kun ohjelma käyttää osoitinta ajonaikana, on sen ensin tarkistettava, että osoittimelle suoritettava operaatio on sallittu. Osoittimelle suoritettavan operaation tarkistus tehdään käyttämällä tietoa osoittimen kantaosoitteesta sekä sen viimeisestä sallitusta osoitteesta. (3, s. 2–3.)



KUVA 3. Paranneltu osoitin (3, s. 2)

2.3.2 Stack Smashing Protector

SSP (Stack Smashing Protector) on GCC:n (GNU Compiler Collection) laajennus, jolla voidaan havaita pinon ylivuotoja. SSP järjestää funktioissa määritellyt osoittimet ennen lokaaleja puskureita sekä siirtää argumenteissa esiintyvät osoittimet lokaalien puskureiden perään muistissa. Lokaalien muuttujien järjestely muistissa suojaa funktiossa määriteltyjä osoittimia puskurin ylivuodolta. SSP lisää funktioon suojamuuttujan, jolla pyritään estämään argumenteissa, funktion paluuosoitteessa ja pino-osoittimessa tapahtuvat muutokset puskurin ylivuodon sattuessa (kuva 4). Suojamuuttuja saa arvokseen satunnaisen arvon, jonka mahdollinen muutos tarkistetaan funktion juoksun aikana. Mikäli muuttujan arvo on muuttunut, voidaan ohjelman suoritus keskeyttää hallitusti sekä informoida kehittäjää ohjelman kohdasta, jossa puskurin ylivuoto on tapahtunut. (1, s. 70–71.)



KUVA 4. SSP-kehyksen struktuuri (1, s. 70–71)

3 SUUNNITTELU

Suojamuuttujien lisäämiseen suunniteltiin ohjelma, joka vastaanottaa kehittäjältä parametreja, joita käyttäen yksittäinen lähdekoodin puskuri suojataan. Lähdekoodin sisältävästä tiedostosta generoidaan suojamuuttujia sisältävä kopio, joka lisätään käännökseen alkuperäisen tiedoston tilalle. Globaalien puskureiden lisäksi lokaaleja puskureja on pystyttävä suojaamaan. Globaalit suojamuuttujat on kyettävä tarkistamaan käskystä ja lokaalien suojamuuttujien tarkistukset suoritetaan funktioissa kehittäjän merkitsemissä kohdissa. Jos tarkistuksessa ilmenee ylivuoto, kutsutaan kehittäjän määrittävää funktiota. Puskuri on kyettävä suojaamaan molemmista päistä muistia.

Globaalit suojamuuttujat sijoittuvat aina datasegmenttiin, koska ne alustetaan määrityksissä. Samasta syystä lokaalit suojamuuttujat sijoittuvat pinosegmenttiin. Suojattava puskuri voi sijoittua BSS-, data- tai pinosegmentteihin, joten lähdekoodissa puskurin viereen määritetty suojamuuttuja ei välttämättä sijaitse muistissa puskurin vieressä. Lisäksi vaikka suojamuuttuja ja puskuri sijoittuisivatkin samaan segmenttiin, voi kääntäjä optimoida puskurin ja suojamuuttujan erilleen muistissa. Puskurille määritetyt suojamuuttujat ja puskuri on strukturoitava, jotta suojamuuttujat sijoittuvat puskurin ympärille muistissa (kuva 5).

Globaalien puskureiden struktuurit täytyy esitellä erillisessä otsikkotiedostossa ja ne ovat muutettava extern-tyyppiseksi, jotta niiden näkyvyys ulottuu suojamuuttujia tarkistavaan tiedostoon. Lokaalien puskureiden struktuurit voidaan esitellä puskureiden määritysvaiheessa, koska viittaukset kyseisiin struktuureihin rajoittuvat suojattavien puskureiden funktioihin. Puskurin koko täytyy määrittää struktuurin esittelyssä. Lähdekoodissa puskurin koko ilmoitetaan joko esittelyssä tai se määräytyy puskurille annettujen arvojen lukumäärästä.

Yksittäinen suojamuuttuja on kyettävä deaktivoimaan ohjelmalla käännöksestä. Lisäksi tiedoston kaikki suojamuuttujat ovat kyettävä deaktivoimaan ohjelmalla deaktivoimalla kyseinen tiedosto ohjelmasta. Kehittäjän syöttämät parametrit suojamuuttujan lisäykselle on tallennettava, jotta suojamuuttuja voidaan lisätä uudestaan toisella istunnolla. Suojamuuttujat indeksoidaan, jotta puskurin ylivuoto voidaan paikantaa. Samaa puskuria suojaavat suojamuuttujat erotellaan toisistaan positionumerolla.

```
struct safeguard_ID_s
{
    #if defined(FILE) && defined(ID) && defined(POS)
    volatile unsigned char safeguard_ID_POS;
    #endif
    int buffer[3];
    #if defined(FILE) && defined(ID) && defined(POS)
    volatile unsigned char safeguard_ID_POS;
    #endif
};

struct safeguard_ID_s safeguard_ID =
{
    #if defined(FILE) && defined(ID) && defined(POS)
    .safeguard_ID_POS = 170,
    #endif
    .buffer = {1,2,3},
    #if defined(FILE) && defined(ID) && defined(POS)
    .safeguard_ID_POS = 170
    #endif
};
```

KUVA 5. Esimerkki struktuurin esittelystä ja määrityksestä

Lähdekoodiin sijoitetaan tunnisteita, joiden avulla paikannetaan suojattava puskuri (kuva 6). Lisäksi lokaalin puskurin tarkistuskutsun sijainti ilmoitetaan tunnisteella. Funktion ympärille lisättävät ylä- ja alatunnisteet ilmoittavat ohjelmalle lokaalin alueen alun ja lopun sekä sen nimen. Funktion ympärille sijoitetut tunnisteet ovat tarpeellisia, koska tiedostossa voi esiintyä useita samannimisiä lokaaleja muuttujia.

```
//<safeguard_local_space='testfunction'>
int testfunction()
{
    int buffer[] = {1,2,3};
    buffer[3] = 4;
    //<safeguard_checkpoint>
    return buffer[0];
}
//</safeguard_local_space='testfunction'>
```

KUVA 6. Esimerkki funktion ympärille lisätyistä tunnisteista

Lähdekoodin globaalit puskurit ovat sijoitettava myös omien tunnisteiden sisälle (kuva 7). Globaalin alueen tunnisteilla varmistetaan, että kehittäjä ei vahingossa suojaa lokaalia puskuria globaalina.

```
//<safeguard_global_space>
int buffer[3];
//</safeguard_global_space>
```

KUVA 7. Esimerkki tunnisteilla ilmoitetusta globaalista alueesta

Globaalien suojamuuttujien tarkistukset generoidaan yhteen funktioon, jossa ne kaikki tarkistetaan käskystä (kuva 8).

```
void global_safeguard_check(void)
{
    if (safeguard_1.safeguard_1_0 != 170)
        // call user function
    if (safeguard_1.safeguard_1_1 != 170)
        // call user function
    .
    .
    .
    if (safeguard_n.safeguard_n_0 != 170)
        // call user function
    if (safeguard_n.safeguard_n_1 != 170)
        // call user function
}
```

KUVA 8. Esimerkki globaalien suojamuuttujien tarkistuksesta

Lokaalien suojamuuttujien tarkistuksiin luodaan funktio, joka vastaanottaa parametreina suojamuuttujan indeksin, position, suojamuuttujan alkuperäisen arvon ja suojamuuttujan ajonaikaisen arvon. Funktiota kutsutaan kehittäjän määrittämissä tarkistuspisteissä. Lokaalin suojamuuttujan funktio voi sisältää useita tarkistuspisteitä. Suojamuuttujat määritellään tietotyypillä unsigned char, joka varaa tavun verran muistia, joten suojamuuttuja voi saada arvokseen 256 eri arvoa. Jos ylivuotava tavu olisi satunnainen, havaittaisiin ylivuoto 255/256:n todennäköisyydellä.

4 AVOIMEN LÄHDEKOODIN KIRJASTOT

Sovellus on riippuvainen kolmesta avoimen lähdekoodin kirjastosta. Kirjastoilla ratkaistaan sovelluksen käytettävyyteen liittyviä haasteita ja toteutetaan tiedonhallinnan asettamat vaatimukset.

4.1 SQLite

SQLite on tietokantamoottori, jonka lähdekoodi on public domain -lisensoitu. Tietokantaa hallitaan kirjoittamalla ja lukemalla kiintomuistissa sijaitsevaa tietokantatiedostoa (4). SQLite ei vaadi erillistä palvelinprosessia tietokannan hallintaan. Palvelinprosessittoman tietokantamoottorin etuja ovat, ettei tietokantaa käyttävän sovelluksen tarvitse suorittaa prosessien välistä kommunikaatiota ja palvelinprosessia ei tarvitse ylläpitää. Toisaalta palvelinprosessia käyttävät tietokantamoottorit suojaavat tietokantaa paremmin korruptiolta ja tietokantaan pääsyä voidaan säädellä tarkemmin. Vaikka SQLitestä puuttuukin perinteinen asiakas-palvelin -arkkitehtuuri, voi tietokantaa käyttää yhtäaikaisesti usea eri sovellus. (5.) SQLite on alustariippumaton, koska se on riippuvainen ainoastaan C-ohjelmointikielen standradikirjastosta (6).

Päätin käyttää tietokantaa sovelluksen tiedonhallintaan, koska tietueiden välille oli muodostettava relaatioita. Valitsin SQLiten tietokantamoottoriksi, koska minulla oli aiempaa kokemusta sen käytöstä ja sen käyttöönotto on yksinkertaista johtuen tietokannan lokaalisuudesta.

4.2 wxWidgets

wxWidgets on sovelluskehitykseen tarkoitettu ohjelmointirajapinta, jolla voidaan toteuttaa graafinen käyttöliittymä sovellukselle usealle eri käyttöjärjestelmälle. wxWidgets-kirjastolla luotu sovellus perii ulkoasunsa käyttöjärjestelmältä, joten sovellus näyttää natiivilta käyttöjärjestelmän sovellukselta. wxWidgets on kirjoitettu C++-ohjelmointikielellä, mutta kirjastoa voidaan käyttää muillakin ohjelmointikielillä, kuten Pythonilla, Perlillä ja C Sharpilla. (7.) Kirjasto on wxWidgets-lisensoitu ja pohjautuu GPL-lisenssiin (8).

Alkuperäinen suunnitelma oli toteuttaa komentorivipohjainen sovellus mutta ymmärsin pian, että jatkuva tietueiden tulostaminen ja navigointi sovelluksessa konsolia käyttäen

olivat sekavaa ja tehotonta (kuva 9). Edellä mainittujen ongelmien takia päädyin toteuttamaan sovellukselle graafisen käyttöliittymän. Valitsin graafisen käyttöliittymän toteutukselle wxWidgets-kirjaston, koska se on avointa lähdekoodia sekä toimii yleisimmillä käyttöjärjestelmillä.

```
>> select file 1
Selected file test_file_0.cpp
>>> insert global safeguard before table0
New global safeguard protecting variable (table0) in file (test_file_0.cpp)
>>> insert global safeguard after table0
New global safeguard protecting variable (table0) in file (test_file_0.cpp)
>>> abort
>> list database files
file                    file id
----------------------------------------
test_file_0.cpp         1
test_file_1.cpp         2
>> select file 2
Selected file test_file_1.cpp
>>> insert global safeguard before table1
New global safeguard protecting variable (table1) in file (test_file_1.cpp)
>>>
```

KUVA 9. Esimerkki alkuperäisen komentorivipohjaisen ohjelman käytöstä

4.3 wxSQLite3

wxSQLite3 on lisäosa wxWidgets-kirjastolle, joka tarjoaa ominaisuuksia SQLite-tietokannan hallintaan (9). wxSQLite3-lisäosa perii lisenssin wxWidgets-kirjastolta (10).

Implementoin wxSQLite3-lisäosan wxWidgets-kirjastoon, koska se yksinkertaistaa lähdekoodia tietokannan kyselyjen ja lisäämisten osalta.

5 TOTEUTUS

Sovellus toteutettiin C++-ohjelmointikielellä ja C++11 -standardilla. Käännöstyökaluna Windows-ympäristössä käytettiin MinGW 5.1.0 -kääntäjää ja Linux-ympäristössä GCC 5.4.0 -kääntäjää. Lähdekoodin versionhallintaan käytettiin Gittiä.

wxWidgets-kirjaston 3.0.3- ja wxSQLite3-lisäosan 3.5.8 -versiosta käännettiin staattiset debug- ja release-käännökset molemmille ympäristöille. SQLite3-kirjasto oli integroituna wxSQLite3-lisäosaan.

Ohjelmointiympäristönä toimi Code::Blocks 16.01, joka tarjosi tuen wxWidgets-projektin luomiseen. Ohjelmointiympäristö tarjosi myös graafisia työkaluja käyttöliittymän toteuttamiselle wxWidgets-kirjastolla, mutta käyttöliittymän lähdekoodi päädyttiin toteuttamaan itse. Käyttöliittymäkomponenteista oli esimerkkejä wxWidgets-kirjaston mukana tulleissa esimerkkiohjelmissa.

5.1 Tietokanta

5.1.1 Tietokannan kuvaus

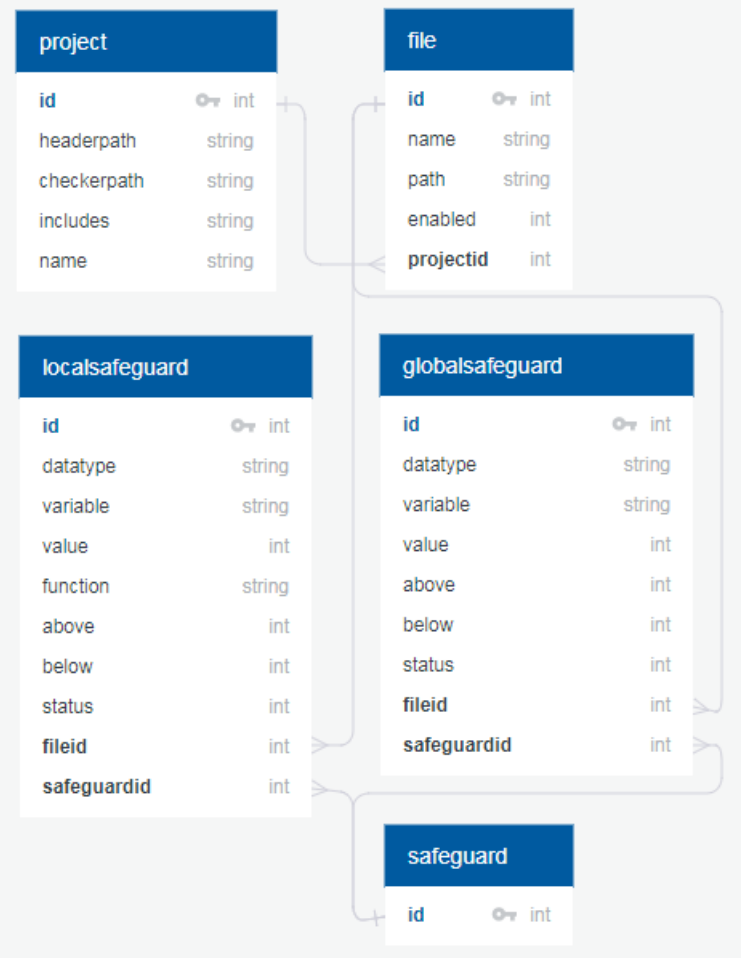
Tietokanta koostuu viidestä taulusta (kuva 10). Tietokantaa ei ole normalisoitu kyselyjen yksinkertaistamiseksi. Taulujen viiteavaimet määritetään ON DELETE CASCADE -tyyppisiksi, jolloin poistettava tietue vyöryttää poiston myös siihen viittaaviin tietueisiin (liite 1). Taulujen id-kentät ovat määritetty pääavaimiksi.

Project-taulun name-kenttä on määritetty uniikiksi, jotta samannimisiä projekteja ei voi esiintyä. Taulussa on kentät tarkistus- ja otsikkotiedoston poluille sekä otsikkotiedoston lisäsisällölle.

File-taulu sisältää kentät tiedoston nimelle ja polulle sekä tiedoston aktivoinnille. Taulu viittaa project-tauluun.

Safeguard-taulun ainut kenttä on se pääavain. Taulun id-kenttää käytetään suojamuuttujien indeksointiin. Lokaalit ja globaalit suojamuuttujat jakavat indeksialtaan.

Globalsafeguard- ja localsafeguard-taulut sisältävät kentät suojattavan puskurin tietotyypille ja nimelle, suojamuuttujan arvolle, ylä- ja alapositioiden aktivoinneille sekä statusnumerolle. Lisäksi localsafeguard-taulu sisältää kentän suojattavan puskurin funktion nimelle. Taulut viittaavat file- ja safeguard-tauluihin.



KUVA 10. Tietokannan relaatiot

5.1.2 Tietokannan käyttö

Lähdekoodissa olevat viittaukset tietokantaan on sijoitettu yhteen tiedostoon, jossa tietokannalle suoritettavat operaatiot ovat sijoitettu omiin funktioihin. Tietokanta avataan sovelluksen käynnistyessä ja suljetaan sovelluksen sammutuksen yhteydessä. Käyttöjärjestelmä estää tietokantatiedoston poistamisen tai siirtämisen sovelluksen ajonaikana. Jokaista tietokannan tietuetta vastaa oma struktuuri sovelluksessa (kuva 11). Tietuestruktuuriin voidaan kysyä tietokannasta yksi tietue ja vastaavasti sen arvot voidaan määrittää

sovelluksessa sekä lisätä tietokantaan. Globalsafeguard- ja file-taulujen tietuestruktuureissa on kommenteilla merkittyjä muuttujia, jotka eivät esiinny tietokannassa kenttinä. Näitä muuttujia käytetään suojamuuttujan tiedostoon lisäämisen yhteydessä.

```
struct SafeguardLocal_s
{
    int id;
    wxString datatype;
    unsigned int value;
    wxString variable;
    wxString func;
    bool above;
    bool below;
    int status;
    int fileId;
    int safeguardId;
};

struct SafeguardGlobal_s
{
    int id;
    wxString datatype;
    unsigned int value;
    wxString variable;
    bool above;
    bool below;
    int status;
    int fileId;
    int safeguardId;
    /* Non database data */
    wxString globalSize;
};
```

```
struct SafeguardFile_s
{
    int id;
    wxString name;
    wxString path;
    bool enabled;
    int projectId;
    /* Non database data */
    int errors;
};

struct SafeguardProject_s
{
    int id;
    wxString headerPath;
    wxString checkerPath;
    wxString name;
    wxString additionalIncludes;
};
```

KUVA 11. Tietokannan tietueita vastaavat struktuurit

Kokonaisille tietokannan tauluille on esiteltynä omat struktuurit (kuva 12). Taulustruktuureissa vektori sisältää tietuestruktuureja. Lisäksi taulustruktuureissa on osoitin, joka osoittaa tietuestruktuuriin, jolla taulustruktuurin tietueet ovat kysytty. Project-taululle ei ole luotu taulustruktuuria, koska taulu ei viittaa mihinkään tauluun.

```
struct SafeguardFileCon_s
{
    std::vector<SafeguardFile_s> files;
    SafeguardProject_s *project;
};

struct SafeguardGlobalCon_s
{
    std::vector<SafeguardGlobal_s> globals;
    SafeguardFile_s *file;
};

struct SafeguardLocalCon_s
{
    std::vector<SafeguardLocal_s> locals;
    SafeguardFile_s *file;
};
```

KUVA 12. Tietokannan tauluja vastaavat struktuurit

Tietokannan hallintaan on luotu funktioita tietueiden lisäämiseen, poistamiseen, päivittämiseen ja kyselyyn (kuva 13). Funktioiden nimien alussa esiintyy suoritettava operaatio ja lopussa kohteena oleva taulu. Globaali ja lokaali suojamuuttuja poistetaan poistamalla safeguard-taulun id-kentän arvo, joka vyöryttää operaation globalsafeguard- ja localsafeguard-tauluihin. Koska project-taululle ei ole luotu taulustruktuuria, palauttaa kysely sen tietuestruktuurit vektorina.

```
std::vector<SafeguardProject_s> GetProjects();
SafeguardFileCon_s GetFiles(SafeguardProject_s *project);
SafeguardGlobalCon_s GetGlobals(SafeguardFile_s *file);
SafeguardLocalCon_s GetLocals(SafeguardFile_s *file);
int InsertProject(SafeguardProject_s *project);
int InsertFile(SafeguardFile_s *file);
int InsertGlobal(SafeguardGlobal_s *global);
int InsertLocal(SafeguardLocal_s *local);
int DeleteProject(SafeguardProject_s *project);
int DeleteFile(SafeguardFile_s *file);
int DeleteSafeguard(int safeguardId);
int UpdateProject(SafeguardProject_s *project);
int UpdateFile(SafeguardFile_s *file);
int UpdateGlobal(SafeguardGlobal_s *global);
int UpdateLocal(SafeguardLocal_s *local);
```

KUVA 13. Perusfunktiot tietokannan käyttöön

Erityisempiä tietokantaan suoritettavia funktioita ovat statusten päivitykset, tietueen olemassaolon tarkistaminen ja uuden suojamuuttujaindeksi luominen (kuva 14). Globaalien ja lokaalien suojamuuttujien statukset voidaan päivittää yksitellen tai massana. Suojamuuttujan status päivitetään sen tiedostoon lisäämisen yhteydessä. Projektit voivat sisältää saman tiedoston, mutta projekti ei voi sisältää samaa tiedostoa useasti. Tiedoston

olemassaolo tarkistetaan kysymällä tietokannasta tiedoston lukumäärää projektin indeksillä. Vastaavasti samoja globaaleja ja lokaaleja suojamuuttujia voi esiintyä tietokannassa, mutta projekti ei voi sisältää samaa suojamuuttujaa useasti. Globaalin suojamuuttujan olemassaolo tarkistetaan kysymällä tietokannasta suojamuuttujan lukumäärää suojattavan puskurin nimellä ja tiedoston indeksillä. Lokaalin suojamuuttujan olemassaolo tarkistetaan globaalin suojamuuttujan tavoin, mutta lisäksi ehtona käytetään funktion nimeä. Projektin indeksiä ei tarvitse käyttää ehtona, koska tiedoston id-kenttä on sidoksessa projektiin. Uuden suojamuuttujaindeksin luomiseksi lisätään safeguard-tauluun tyhjä arvo id-kenttään, jolloin taulu generoi automaattisesti kentän arvon.

```
int MassUpdateGlobalStatus(SafeguardFile_s *file, int status);
int UpdateGlobalStatus(SafeguardGlobal_s *global, int status);
int MassUpdateLocalStatus(SafeguardFile_s *file, int status,
                          wxString func=wxEmptyString);
int UpdateLocalStatus(SafeguardLocal_s *local, int status);
int FileExists(SafeguardFile_s *file);
int GlobalExists(SafeguardGlobal_s *global);
int LocalExists(SafeguardLocal_s *local);
int GetSafeguardId();
```

KUVA 14. Lisäfunktiot tietokannan käyttöön

Tietokantaa käytetään wxSQLite3-kirjaston luokasta wxSQLite3Database muodostetulla oliolla (kuva 15). Tietokannan avauksen yhteydessä tarkistetaan tietokantatiedoston olemassaolo. Sovellus etsii tietokantatiedostoa suoritettavasta kansiosta. Oletuksena SQLite-tietokantamoottori ei huomioi viiteavaimia, joten viiteavaimien käyttö tulee kytkeä erikseen päälle.

```
int OpenDatabase(const wxString &dbName)
{
    int ret = -1;
    wxString query = wxT("PRAGMA foreign_keys = ON;");

    try
    {
        if (wxFileExists(dbName))
        {
            db = new wxSQLite3Database();
            db->Open(dbName);
            db->ExecuteUpdate(query);
            ret = 0;
        }
        else
        {
            wxMessageBox("Database \"safeguard.db\" doesn't exist",
                         wxT("Error"), wxICON_ERROR);
        }
    }
    catch (wxSQLite3Exception &e)
    {
        wxMessageBox(e.GetMessage(), wxT("Error"), wxICON_ERROR);
    }

    return ret;
}
```

KUVA 15. Tietokannan avaus

Olion varaama muisti vapautetaan tietokannan sulkemisen yhteydessä (kuva 16).

```
void CloseDatabase()
{
    try
    {
        if (db != NULL)
        {
            db->Close();
            delete db;
        }
    }
    catch (wxSQLite3Exception &e)
    {
        wxMessageBox(e.GetMessage(), wxT("Error"), wxICON_ERROR);
    }
}
```

KUVA 16. Tietokannan sulkeminen

Tietueen lisääminen, päivitys ja poisto suoritetaan wxSQLite3Database-luokan ExecuteUpdate-funktiolla (kuva 17).

```
int DeleteProject(SafeguardProject_s *project)
{
    int ret = 0;
    wxString query = wxString::Format(wxT("DELETE FROM project WHERE id=%d"),
                                      project->id);

    try
    {
        db->ExecuteUpdate(query);
    }
    catch (wxSQLite3Exception &e)
    {
        ret = -1;
        wxMessageBox(e.GetMessage(), wxT("Error"), wxICON_ERROR);
    }

    return ret;
}
```

KUVA 17. ExecuteUpdate-funktion käyttö

Tietueiden kysely suoritetaan wxSQLite3Database-luokan ExecuteQuery-funktiolla, joka palauttaa wxSQLite3ResultSet-luokan olion (kuva 18). Olio sisältää kyselyn tuloksen ja sitä iteroimalla voidaan tietue kerralla palauttaa kenttien arvoja. Yksittäisen kentän arvoa haetaan oliosta kentän nimellä. Lopuksi olion on kutsuttava Finalize-funktiota, joka vapauttaa tietueiden varaaman muistin.

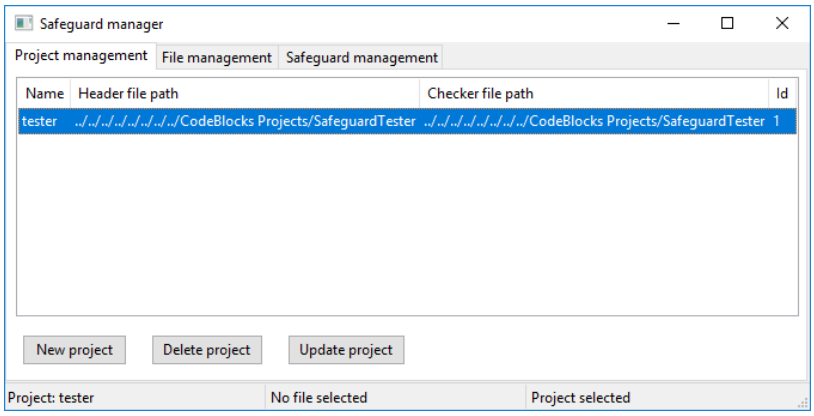
```
SafeguardLocalCon_s GetLocals(SafeguardFile_s *file)
{
    SafeguardLocalCon_s locals;
    locals.file = file;
    wxString query = wxString::Format(wxT("SELECT * FROM localsafeguard WHERE \
                                          fileid=%d"), file->id);

    try
    {
        wxSQLite3ResultSet result = db->ExecuteQuery(query);
        while (result.NextRow())
        {
            SafeguardLocal_s local;
            local.id = result.GetInt(wxT("id"));
            local.datatype = result.GetAsString(wxT("datatype"));
            local.variable = result.GetAsString(wxT("variable"));
            local.value = result.GetInt(wxT("value"));
            local.func = result.GetAsString(wxT("function"));
            local.above = result.GetBool(wxT("above"));
            local.below = result.GetBool(wxT("below"));
            local.status = result.GetInt(wxT("status"));
            local.fileId = result.GetInt(wxT("fileid"));
            local.safeguardId = result.GetInt(wxT("safeguardid"));
            locals.locals.push_back(local);
        }

        result.Finalize();
    }
    catch (wxSQLite3Exception &e)
    {
        wxMessageBox(e.GetMessage(), wxT("Error"), wxICON_ERROR);
    }

    return locals;
}
```

KUVA 18. ExecuteQuery-funktion käyttö

5.2 Käyttöliittymä

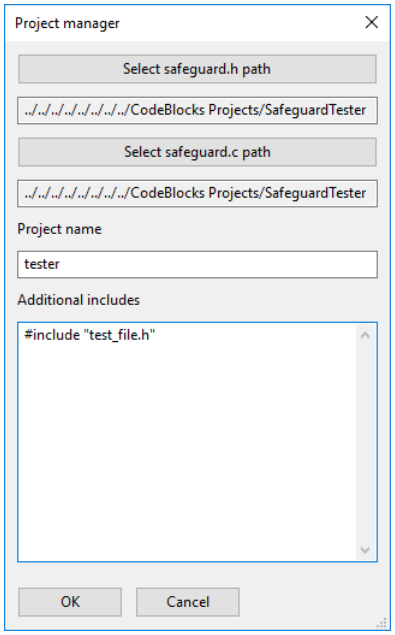
Sovellukselle toteutettiin graafinen käyttöliittymä suojamuuttujien hallinnan yksinkertaistamiseksi.

Käyttöliittymä koostuu kolmesta päänäkymästä ja kolmesta alinäkymästä. Projektinäkymässä on listaus kehittäjän luomista projekteista (kuva 19). Listassa näkyvät projektin nimi, otsikkotiedoston ja tarkistustiedoston sijainti sekä projektin indeksi. Projekti valitaan klikkaamalla listan riviä. Kun projekti on valittu, voidaan sen arvoja päivittää tai se voidaan poistaa tietokannasta.



KUVA 19. Sovelluksen projektinäkymä

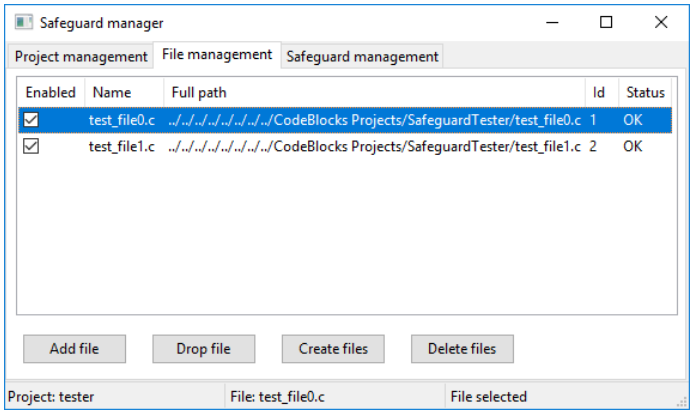
Uuden projektin luomisessa aukeaa alinäkymä (kuva 20). Sama alinäkymä aukeaa projektin päivityksessä ja alinäkymän kentät täytetään valitun projektin arvoilla. Alinäkymän kaksi ylintä painiketta avaavat tiedostonäkymän, jolla valitaan kohdekansio otsikkotiedostolle tai tarkistustiedostolle. Tekstikentissä näkyvät kansioiden polut, joita kehittäjän ei ole mahdollista muokata manuaalisesti. Alinäkymän alimpaan tekstikenttään voidaan tarvittaessa lisätä lisäsisältöä otsikkotiedostolle.



KUVA 20. Alinäkymä uuden projektin lisäykselle

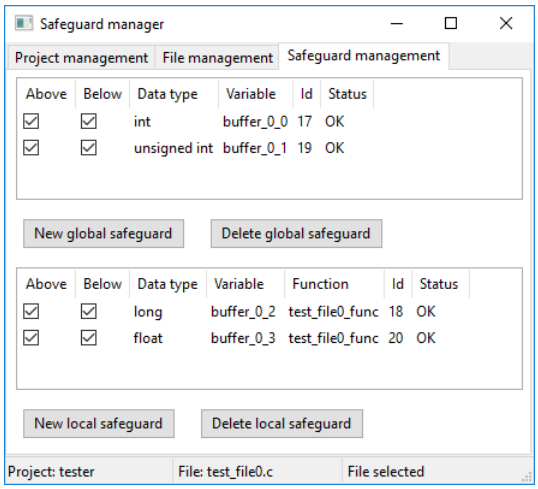
Tiedostonäkymään listautuu valitun projektin tiedostot (kuva 21). Listassa näkyy tiedoston nimi, polku, indeksi ja status. Tiedosto voidaan aktivoida tai deaktivoida klikkaamalla valintaruutua. Tiedosto valitaan klikkaamalla listan riviä. Kun tiedosto on valittu, voidaan

se poistaa tietokannasta. Tiedostonäkymässä voidaan generoida tiedostostoista suojamuuttujia sisältävät kopiot sekä tarvittavat otsikko- ja tarkistustiedostot. Vastaavasti kaikki sovelluksen luomat tiedostot voidaan poistaa. Uuden tiedoston lisäyksessä aukeaa käyttöjärjestelmästä peritty tiedostodialogi, josta kehittäjä valitsee tiedoston.



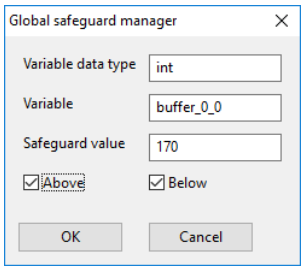
KUVA 21. Sovelluksen tiedostonäkymä

Suojamuuttujanäkymä sisältää listaukset valitun tiedoston globaaleista ja lokaaleista suojamuuttujista (kuva 22). Globaalien suojamuuttujien listassa näkyvät suojattavan muuttujan tietotyyppi ja nimi sekä suojamuuttujan indeksi ja status. Lokaalien suojamuuttujien lista vastaa globaalien suojamuuttujien listaa mutta lisäksi näkyvissä on funktion nimi, jossa suojattava muuttuja sijaitsee. Suojamuuttuja voidaan aktivoida tai deaktivoida klikkaamalla positioiden valintaruutuja.



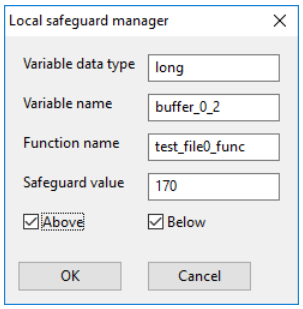
KUVA 22. Sovelluksen suojamuuttujanäkymä

Lisättäessä uusi globaali suojamuuttuja aukeaa alinäkymä (kuva 23). Kehittäjä kirjoittaa tekstikenttiin suojattavan puskurin tietotyypin ja nimen sekä suojamuuttujan arvon. Suojamuuttuja voidaan aktivoida tai deaktivoida sen luomisen yhteydessä.



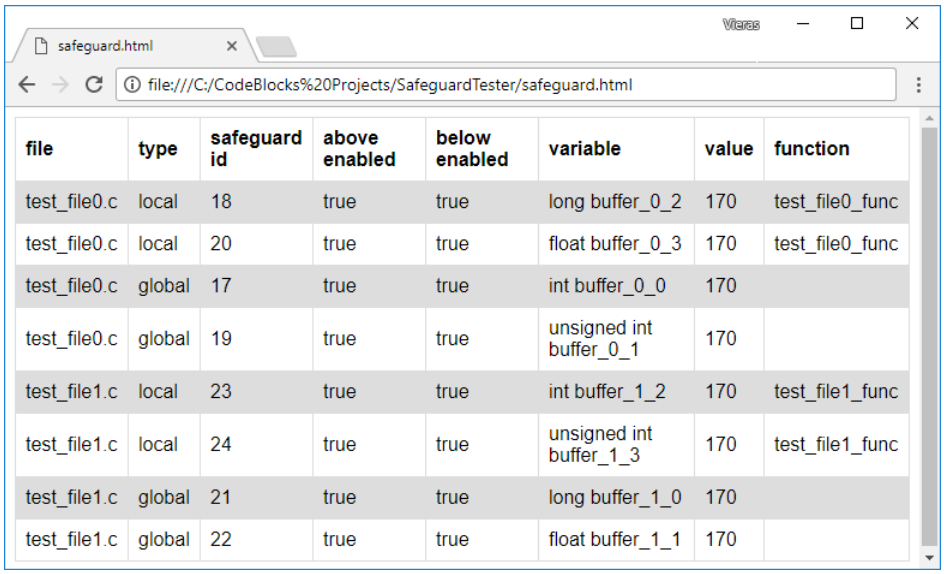
KUVA 23. Alinäkymä globaalin suojamuuttujan lisäykselle

Lokaalisuojamuuttujan lisääminen vastaa globaalisuojamuuttujan lisäämistä mutta kehittäjän on ilmoitettava suojattavan puskurin funktion nimi (kuva 24).



KUVA 24. Alinäkymä lokaalin suojamuuttujan lisäykselle

Suojamuuttujista generoituu taulukko HTML-tiedostoon, joka sijaitsee tarkistustiedoston kansiossa (kuva 25). Sovelluksessa listat suojamuuttujista ovat tiedostokohtaisia, joten yksittäisen suojamuuttujan löytäminen on nopeampaa taulukkoa käyttäen.



file	type	safeguard id	above enabled	below enabled	variable	value	function
test_file0.c	local	18	true	true	long buffer_0_2	170	test_file0_func
test_file0.c	local	20	true	true	float buffer_0_3	170	test_file0_func
test_file0.c	global	17	true	true	int buffer_0_0	170	
test_file0.c	global	19	true	true	unsigned int buffer_0_1	170	
test_file1.c	local	23	true	true	int buffer_1_2	170	test_file1_func
test_file1.c	local	24	true	true	unsigned int buffer_1_3	170	test_file1_func
test_file1.c	global	21	true	true	long buffer_1_0	170	
test_file1.c	global	22	true	true	float buffer_1_1	170	

KUVA 25. HTML-tiedoston taulukko generoituneista suojamuuttujista

5.3 Säännöllinen lauseke

Säännöllinen lauseke on malli, joka täsmää määrättyihin merkkijonoihin. Sulkumerkeillä määritetään lausekkeen osa. Osia voi olla useita ja niitä käytetään erittelemään tietoa kohteena olevasta merkkijonosta. Pakoluokka ”\s” etsii kohteesta tyhjiä merkkejä (esim. tabulaattori tai välilyönti). Plusmerkki laajentaa etsinnän tuottamaan yhden tai useamman tuloksen, kun taas jokerimerkki laajentaa etsinnän tuottamaan useamman tuloksen tai ei tulosta ollenkaan. Hakasulkeilla määritetään sallittuja merkkejä. Kysymysmerkillä ehdollisestaan tulos; tulos voi löytyä kerran tai ei kertaakaan. (11.)

Puskurin etsimiseen lähdekoodista käytettävä säännöllinen lauseke on esiteltynä kuvassa 26. Lauseke aloitetaan ehdollisella osalla, johon etsitään merkkijonoa ”static” ja yhtä tai useampaa tyhjää merkkiä. Käyttämällä wxString-luokan Format-funktiota, voidaan merkkijonon sisältämiin ”%s” -merkkeihin lisätä toinen merkkijono. Ensimmäiseen ”%s” -merkkiin lisätään suojattavan puskurin tietotyyppi ja toiseen puskurin nimi. Tietotyypin ja nimen välissä on oltava vähintään yksi tyhjä merkki. Puskurin nimen jälkeen etsitään hakasulkeita, joiden sisällä määritellään lausekkeen toinen osa. Osan sallittuja merkkejä ovat pienet sekä suuret englanninkielen aakkoset, numerot, alaviiva ja tyhjät merkit. Sallittuja merkkejä voi esiintyä osassa useita tai ei yhtään. Jos osaan tallentuu merkkejä, on puskurin koko ilmoitettu. Lausekkeen kolmas osa on ehdollinen ja sillä etsitään puskurille annettuja arvoja. Osa aloitetaan yhtäläisyysmerkin etsinnällä, jota seuraa aaltosulkeet. Aaltosulkeiden sisällä määritetään samat sallitut merkit kuin toisessa

osassa ja lisäksi piste sekä pilkku sallitaan. Lausekkeen viimeinen etsittävä merkki on puolipiste.

```
"(static\\s+)?%s\\s+%s\\s*\\[\\s*([a-zA-Z0-9_\\s]*)\\s*\\]\\s*(=\\s*{\\s*[a-zA-Z0-9_',\\.\\s]*})?\\s*;"
```

KUVA 26. Säännöllinen lauseke puskurin etsimiseen

Kaikkien lausekkeen termien välissä sallitaan tyhjiä merkkejä, joten kuvan 27 mukainen puskurin määritys kyetään löytämään lähdekoodista.

```
static    int
buffer [
            ]
        =
           {
           1
2,     3
        };
```

KUVA 27. Esimerkki huonon tavan mukaisesta muuttujan määrityksestä

Lokaalin alueen etsimiseen käytettävä säännöllinen lauseke on kuvassa 28. Lausekkeen ensimmäiseen merkkiin ”%s” sijoitetaan kauttamerkki tai tyhjä merkkijono, jolloin lausekkeen formaatti toimii molempien ylä- ja alatunnisteiden lausekkeina. Etsittäessä täsmällistä lokaalia aluetta sijoitetaan lausekkeen toiseen ”%s” -merkkiin kehittäjän ilmoittama suojattavan puskurin funktion nimi. Tiedoston kaikki lokaalit alueet kyetään löytämään, kun merkkiin sijoitetaan merkkijono ”(.*)”, joka on säännöllisen lausekkeen syntaksia. Tämä merkkijono luo lausekkeen osan, johon tallentuu heittomerkkien välinen sisältö.

```
"\\/\\/<%ssafeguard_local_space='%s'>"
```

KUVA 28. Säännöllinen lauseke lokaalin alueen etsimiseen

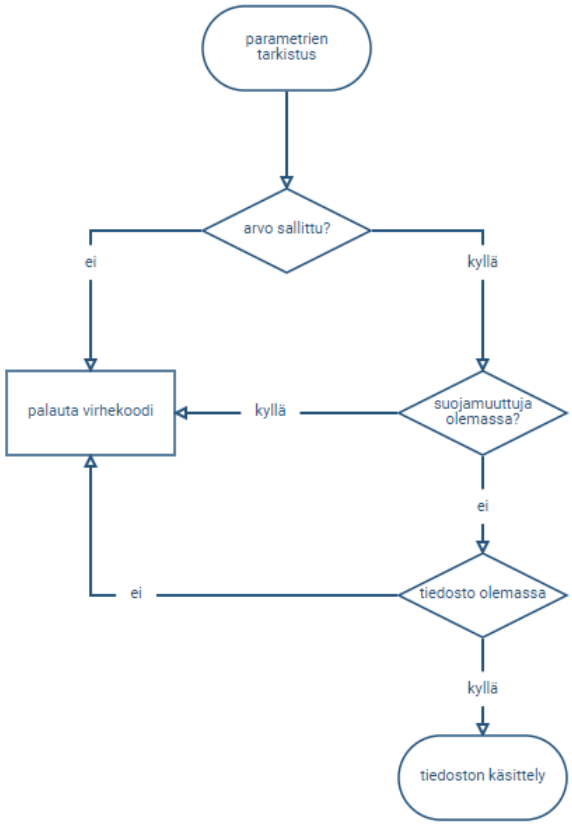
Puskureiden uudelleennimeämiset suoritetaan kuvan 29 mukaisella säännöllisellä lausekkeella. Lausekkeen ”%s” -merkkiin sijoitetaan puskurin nimi, jonka ympärille asetetaan säännöllisen lausekkeen ”\y” -merkit. Asettamalla kyseiset merkit merkkijonon ympärille etsitään kohteesta kokonaista sanaa.

```
"\\y%s\\y"
```

KUVA 29. Säännöllinen lauseke uudelleennimeämiselle

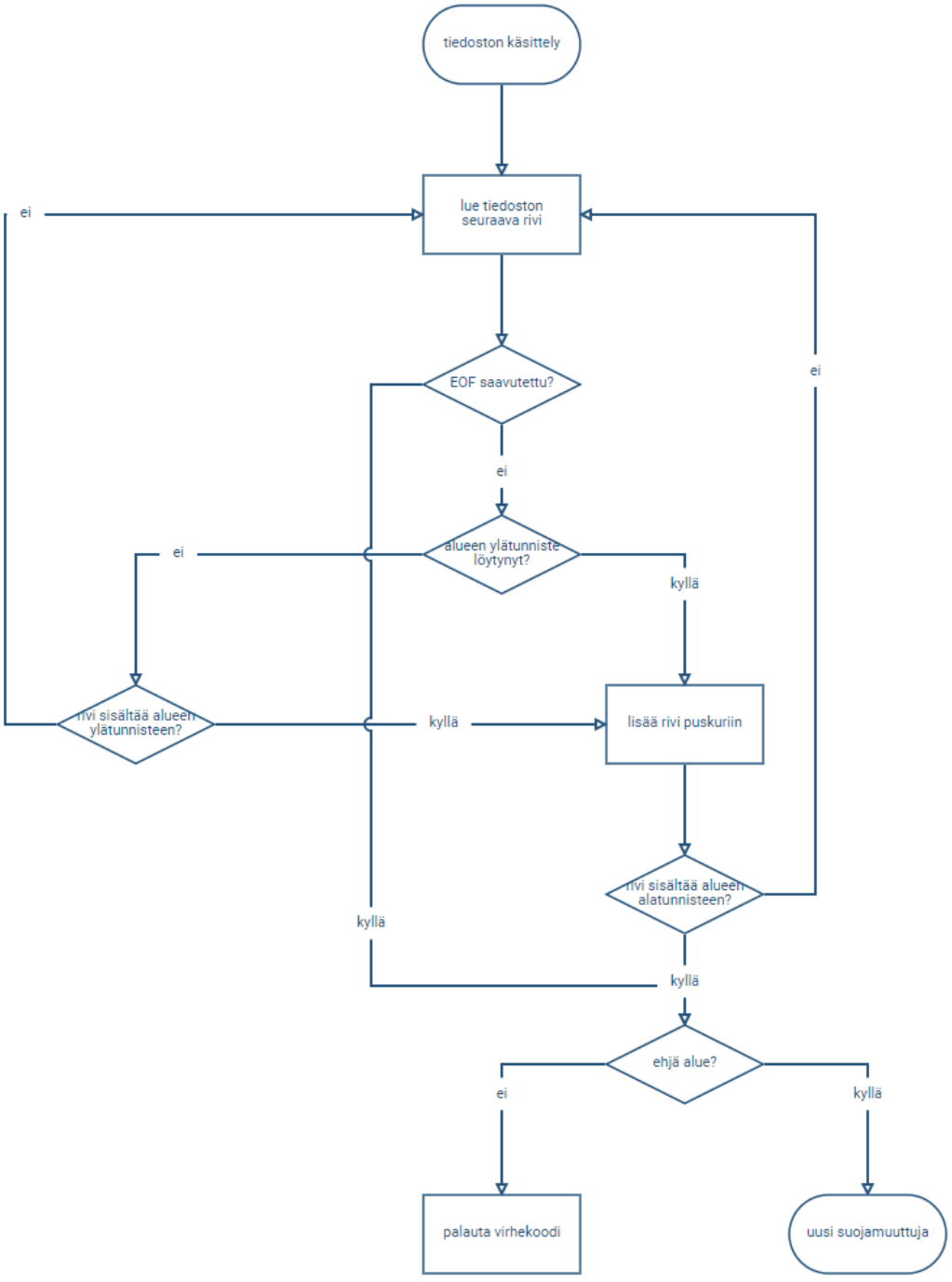
5.4 Suojamuuttujien lisääminen tietokantaan

Ennen suojamuuttujan lisäämistä tietokantaan on tarkistettava kehittäjän syöttämät parametrit (kuva 30). Jos parametri ei ole sallittu, funktio palauttaa virhekoodin ja tulostaa kehittäjälle virheen syyn. Koska suojamuuttuja on unsigned char -tyyppiä, on sen arvon oltava 0–255 välillä. Projekti ei voi sisältää kahta samaa suojamuuttujaa, joten suojamuuttujan olemassaolo tarkistetaan. Jos tiedostoa ei ole olemassa tai sitä ei muusta syystä saada auki, ei suojattavan muuttujan olemassaoloa voida tarkistaa, joten suojamuuttujan lisääminen tietokantaan on keskeytettävä.



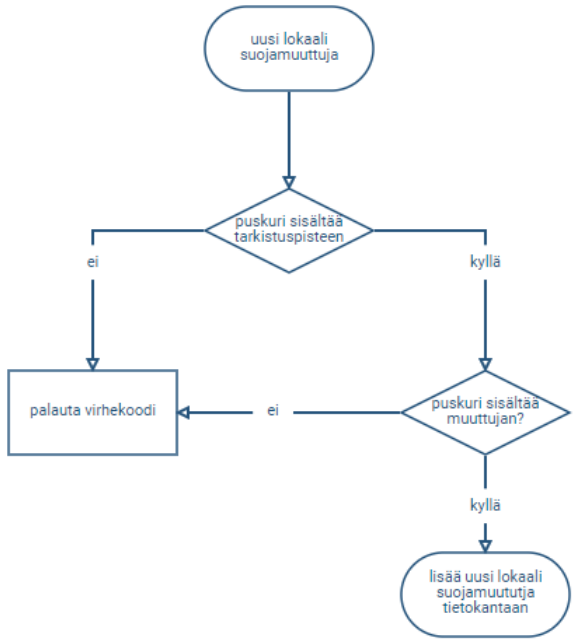
KUVA 30. Vuokaavio parametrien tarkistuksista

Jos tiedosto saadaan auki, luetaan sitä rivi kerralla, kunnes saavutetaan tiedoston viimeinen rivi (EOF) tai suojattavan muuttujan alue on luettuna puskurissa (kuva 31). Rivi lisätään puskuriin ainoastaan, jos se sisältää tunnisteen tai sen sisältö sijaitsee tunnisteiden sisällä. Puskuriin luettu alue on ehjä, kun se sisältää alueen ylä- ja alatunnisteet. Jos alue ei ole ehjä, on kehittäjän lisättävä tunnisteet lähdekoodiin tai tarkistettava tunnisteiden eheys. Lokaalisuojamuuttujien alue on suojattavan muuttujan funktio (kuva 6) ja globaalisuojamuuttujien alue on kehittäjän määrittämä (kuva 7).



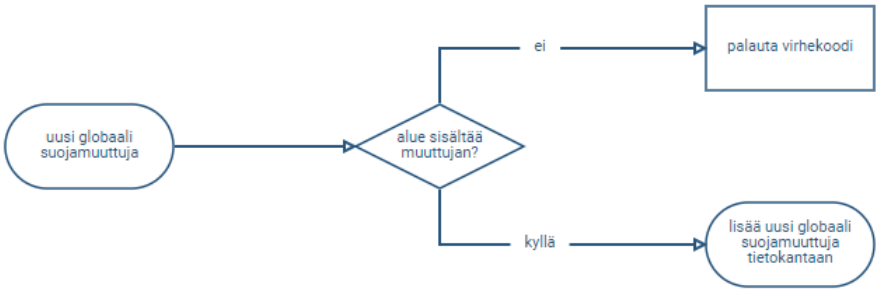
KUVA 31. Vuokaavio tiedoston käsittelystä

Jos lokaalisuojamuuttujan alue on ehjä, tarkistetaan tarkistuspisteen olemassaolo puskurista (kuva 32). Tarkistuspisteitä on oltava vähintään yksi. Lopuksi tarkistetaan itse suojattavan puskurin olemassaolo puskurista. Jos edellä mainitut tarkistukset eivät palauta virhekoodia, uusi lokaalisuojamuuttuja lisätään tietokantaan.



KUVA 32. Vuokaavio lokaalin suojamuuttujan tietokantaan lisäyksestä

Globaalin suojamuuttujan alueen ollessa ehjä tarkistetaan suojattavan puskurin olemassaolo puskurista (kuva 33). Jos suojattava muuttuja löytyy, uusi globaali suojamuuttuja lisätään tietokantaan.

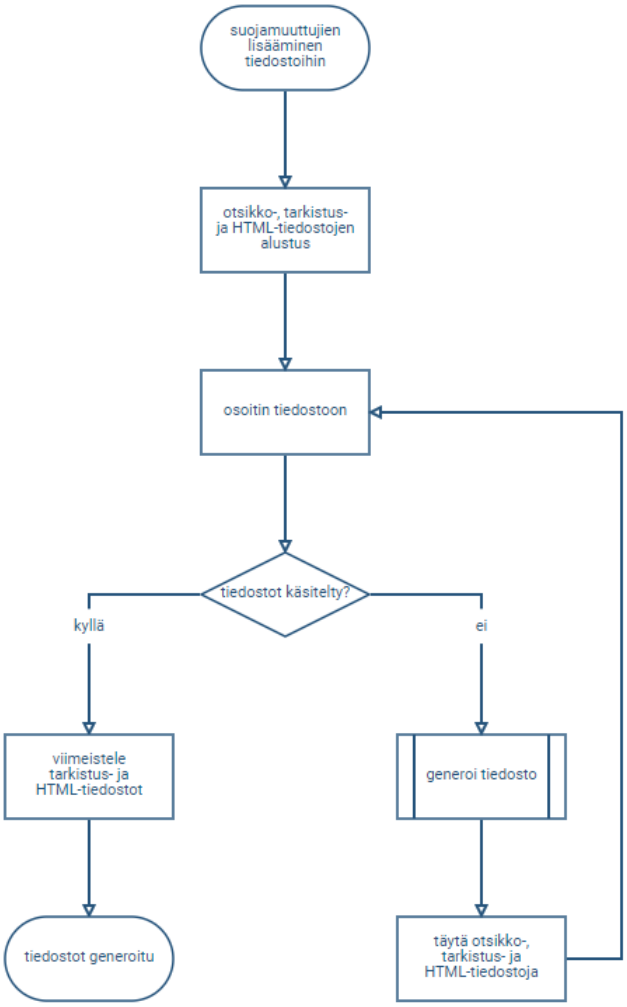


KUVA 33. Vuokaavio globaalin suojamuuttujan tietokantaan lisäyksestä

5.5 Suojamuuttujien lisääminen lähdekoodiin

Suojamuuttujien lisääminen tiedostoihin aloitetaan alustamalla puskurit otsikko-, tarkistus- ja HTML-tiedostoille (kuva 34). Otsikkotiedoston puskurin alkuun lisätään tarkistusfunktioiden esittelyt sekä otsikkotiedostolle määritelty lisäsisältö. Tarkistustiedoston puskurin alkuun määritetään lokaalien suojamuuttujien tarkistusfunktio kokonaisuudessaan sekä globaalien suojamuuttujien tarkistusfunktio sulkematta kuitenkaan funktiota. HTML-tiedoston puskuriin lisätään taulukon otsikot eikä taulukkoa myöskään suljeta. Projektin

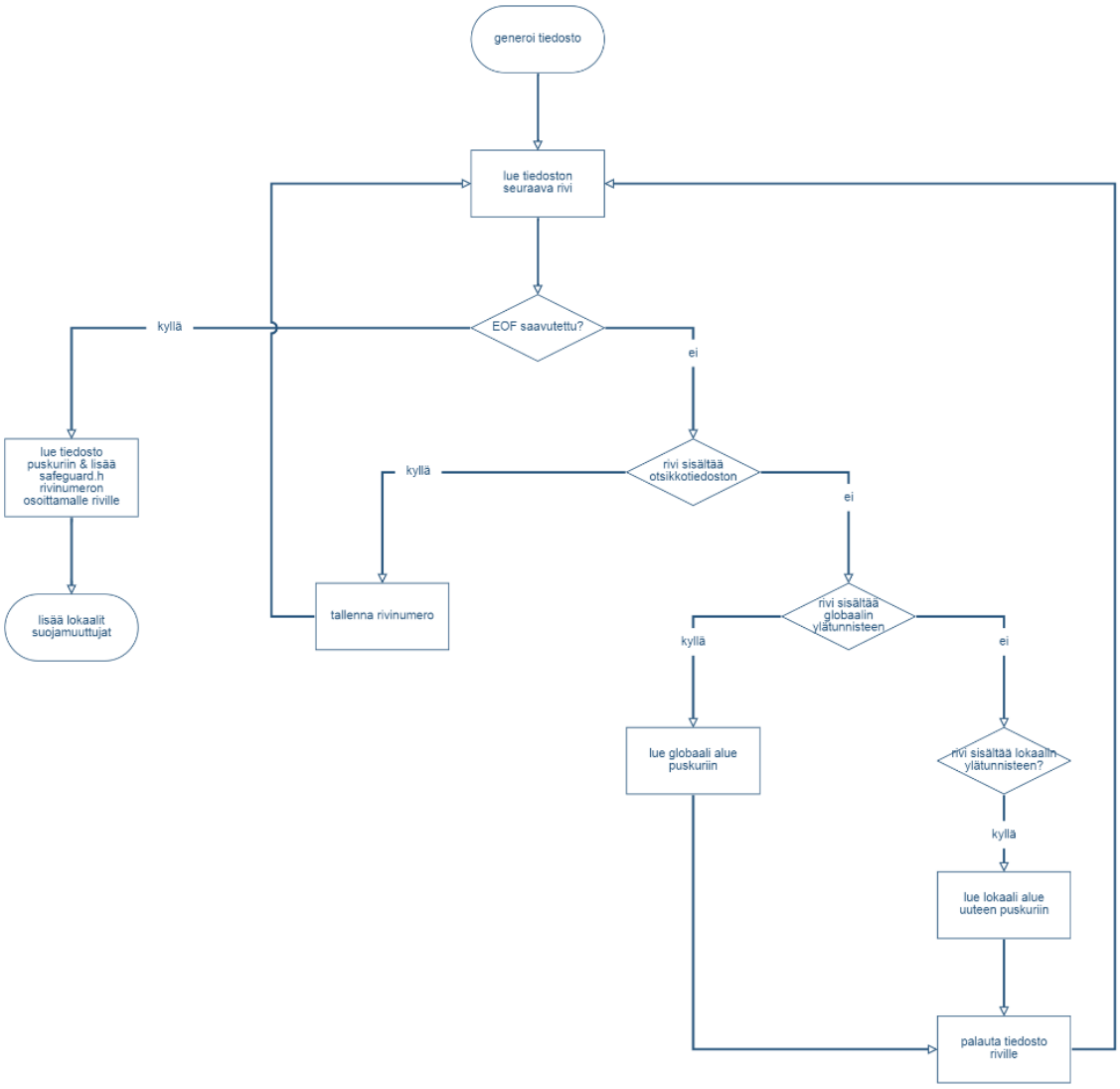
tiedostoja iteroidaan ja tiedosto välitetään funktioon, jossa siitä generoidaan suojamuuttujia sisältävä kopio. Jokaisesta iteroitavasta tiedostosta generoidaan sisältöä otsikko-, tarkistus- ja HTML-tiedostojen puskureihin. Otsikkotiedoston puskuriin kirjoitetaan globaalien suojamuuttujien struktuurien esittelyt sekä suojamuuttujien define-direktiivit. Suojamuuttujan define-direktiivi jätetään kirjoittamatta, mikäli suojamuuttuja on deaktivoitu. Tarkistustiedoston puskuriin kirjoitetaan globaalien suojamuuttujien tarkistuksia aukinaiseen tarkistusfunktioon. HTML-tiedoston puskurin aukinaista HTML-taulukkoa täytetään lisättyjen suojamuuttujien tiedoilla.



KUVA 34. Vuokaavio tiedostojen generoinnista

Tiedoston generointi aloitetaan lukemalla tiedoston jokainen tunnistettavissa oleva alue puskureihin (kuva 35). Globaalille alueelle varataan yksi puskuri ja lokaalien alueiden puskurit työnnetään vektoriin. Rivin sisältäessä alueen ylätunnisteen luetaan tiedostoa

puskuriin, kunnes rivi sisältää alueen alatunnisteen. Jos alueen alatunnistetta ei ole olemassa, luetaan tiedosto loppuun eikä lokaalin alueen puskuria työnnetä vektoriin. Tiedoston riviosoitin palautetaan osoittamaan alueen ylätunnistetta alueen lukemisen jälkeen. Jos rivi sisältää include-direktiivin, tallennetaan rivinumero muuttujaan. Kun tiedoston rivit ovat loppu, kopioidaan tiedoston sisältö puskuriin. Kopioidun tiedoston sisältöön lisätään include-direktiivi tallennetun rivinumeron kohdalle sisällyttämään lähdekoodin tarvitsema otsikkotiedosto.



KUVA 35. Vuokaavio suojamuuttujien lisäyksistä tiedostoon

Jos suojamuuttujan lisäyksessä tapahtuu virhe, päivitetään suojamuuttujan status tietokantaan (kuva 36). Lokaalit suojamuuttujat muodostetaan suorittamalla sisäkkäistä iterointia, jossa ulkotasolla ovat tiedostosta löydetyt lokaalit alueet ja sisätasolla tiedostolle

määritetyt lokaalit suojamuuttujat. Lokaalien suojamuuttujien statukset päivitetään oletuksena arvoon 1, jotta iteroinnin jälkeen tiedetään, mille suojamuuttujalle ei löydetty aluetta.

```
enum SafeguardStatus_e
{
    SAFEGUARD_STATUS_OK = 0,
    SAFEGUARD_STATUS_FUNCTION_NOT_FOUND,
    SAFEGUARD_STATUS_CHECKPOINT_NOT_FOUND,
    SAFEGUARD_STATUS_VARIABLE_NOT_FOUND,
    SAFEGUARD_STATUS_VARIABLE_INVALID_DECLARATION,
    SAFEGUARD_STATUS_CHECK_FUNCTION_CREATE_FAILURE,
    SAFEGUARD_STATUS_SAFEGUARD_CREATE_FAILURE,
    SAFEGUARD_STATUS_FILE_MODIFICATION_FAILURE,
    SAFEGUARD_STATUS_GLOBAL_SPACE_NOT_FOUND,
};
```

KUVA 36. Suojamuuttujan mahdolliset statukset

Jos lokaalin suojamuuttujan alue täsmää tiedostosta löydetyn lokaalin alueen kanssa, tarkistetaan lokaalista alueesta suojattavan puskurin ja tarkistuspisteen olemassaolot. Globaalit suojamuuttujat muodostetaan iteroimalla suojamuuttujia, joiden suojattavien puskureiden olemassaolot tarkistetaan tiedostosta löydetystä globaalista alueesta.

Suojattavien puskureiden koko voidaan määrittää säännöllisen lausekkeen osalla (kuva 26). Jos puskurin kokoa ei ole ilmoitettu, lasketaan puskurille määritettyjen arvojen lukumäärä, joka määrittää muuttujan koon. Suojattavan globaalin puskurin koko on tallennettava SafeguardGlobal_s-tietuestruktuuriin, koska globaalin suojamuuttujan struktuuri täytyy esitellä suojamuuttujien otsikkotiedostossa. Otsikkotiedoston sisältöä kirjoitetaan tiedoston generoinnin jälkeen.

Suojamuuttujat generoidaan puskureissa oleviin alueisiin, jotka korvaavat puskuriin luetun tiedoston alueet. Lokaaleissa alueissa olevat viittaukset suojattuihin lokaaleihin puskureihin muutetaan vastaamaan lokaalien suojamuuttujien struktuureissa määriteltyjä puskureita. Kun puskuriin luetun tiedoston alueet ovat korvattu suojamuuttujia sisältävillä alueilla, lähdekoodin viittaukset globaaleihin suojattuihin puskureihin muutetaan vastaamaan globaalien suojamuuttujien struktuureissa määriteltyjä puskureita.

Suojamuuttujia sisältävä tiedosto luodaan alkuperäisen tiedoston rinnalle, joka saa nimeksi alkuperäisen tiedoston nimen sekä merkkijonon ”_safe” nimen perään. Jos tiedoston suojamuuttujaa ei kyetty lisäämään, nostetaan kyseisen tiedoston SafeguardFile_s-

tietuestruktuurin error-lippu. Kun kaikki tiedostot on iteroitu, suljetaan HTML-tiedoston tagit sekä tarkistustiedoston globaalien suojamuuttujien tarkistusfunktio merkillä ”}”. Lopuksi otsikko-, tarkistus- ja HTML-tiedostot kirjoitetaan kehittäjän määrittämiin kansioihin.

6 TESTAUS

Sovellukselle tehtiin yksikkö- ja integraatiotestausta sovelluskehityksen aikana. Sovelluksen komponentteja testattiin yksin sekä yhdessä. Päädyimme järjestelmätestaamaan valmiin sovelluksen ennen sen käyttöönottoa. Järjestelmätestauksen tarkoituksena on testata sovellukselle asetetut vaatimukset sekä demonstroida suojamuuttujien toimintaa.

6.1 Testaussuunnitelma

Testaukseen luodaan testiprojekti C-ohjelmointikielellä, joka sisältää kaksi testitiedostoa, main-tiedoston ja otsikkotiedoston. Molemmat testitiedostot sisältävät kaksi globaalia ja lokaalia puskuria. Toinen globaaleista ja lokaaleista puskureista on oltava staattinen. Testitiedostoihin luodaan funktiot, joissa puskureille suoritetaan operaatioita nimenmuunnoksen testaamiseksi. Funktioissa aiheutetaan puskureille ylivuotoja varatun muistin molempiin päihin. Puskurit tulee määrittää seuraavalla tavalla:

- Puskurin koko on ilmoitettu ja puskurille ei ole annettu arvoja.
- Puskurin kokoa ei ole ilmoitettu ja puskurille on annettu arvoja.
- Puskurin koko on ilmoitettu define-direktiivillä ja puskurille ei ole annettu arvoja.
- Puskurin kokoa ei ole ilmoitettu ja puskurille annettu define-direktiivisiä arvoja.

Jokainen luettelon puskureiden määritysten malli tulee testata globaali ja lokaali alueilla. Otsikkotiedosto sisältää puskureissa käytettävät define-direktiivit. Main-tiedosto kutsuu testitiedostojen funktioita ja määrittää tarkistusfunktion, jota tarkistustiedoston tarkistusfunktiot kutsuvat. Sovellus todetaan toimivaksi, kun testiprojektin lähdekoodi kääntyy sekä jokainen aiheutettu ylivuoto havaitaan. Staattisten lokaalien puskureiden tulee säilyä staattisina toisin kuin globaalien puskureiden.

6.2 Testauksen toteutus

Liitteet 1 ja 2 sisältävät testaussuunnitelmassa määritettyjen testitiedostojen lähdekoodit. Testitiedostojen funktioissa suoritettava logiikka ennen tarkoituksellisia ylivuotoja ei ole oleellista, koska sillä testataan ainoastaan nimenmuunnosta. Jos nimenmuunnos epäonnistuu, ei tiedosto käänny. Otsikkotiedostossa määritetään define-direktiivejä sekä esitellään funktiot main-tiedostolle (kuva 37).

```
#ifndef TEST_FILE_H_INCLUDED
#define TEST_FILE_H_INCLUDED

#define TEST_SIZE 7
#define TEST_VALUE 9
#define TEST_OVERFLOW

void test_file0_func(void);
void test_file1_func(void);

#endif // TEST_FILE_H_INCLUDED
```

KUVA 37. Testiprojektin otsikkotiedosto

Main-tiedoston tarkistusfunktio tulostaa suojamuuttujan indeksin, position, alkuperäisen arvon sekä puskurin ylivuodon aiheuttaman arvon (kuva 38). Main-funktio kutsuu lopuksi globaalien suojamuuttujien tarkistusfunktiota.

```
#include <stdio.h>
#include "test_file.h"

void safeguard_user_function(unsigned char overval, int pos, int id, unsigned char val)
{
    printf("id: %d, pos: %d, val: 0x%.2X, overval: 0x%.2X\n", id, pos, val, overval);
}

int main()
{
    test_file0_func();
    test_file1_func();
    global_safeguard_check();
    return 0;
}
```

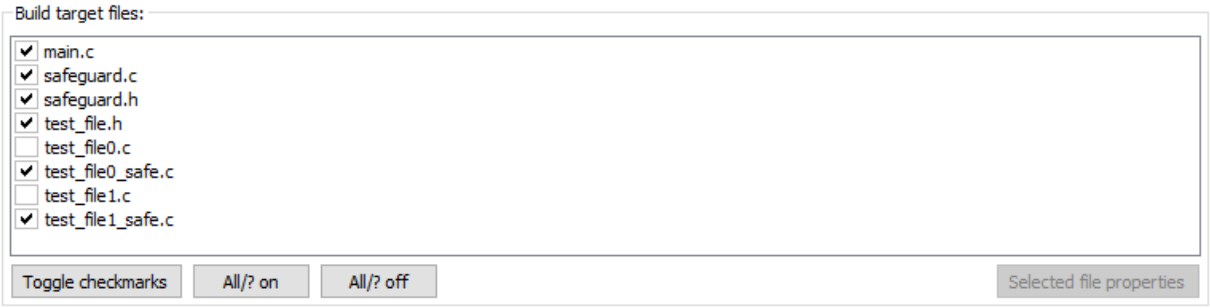
KUVA 38. Testiprojektin main-tiedosto

Sovelluksella luotiin uusi projekti ja polut asetettiin testiprojektin kansioon (kuva 20). Projektiin lisättiin testiprojektin testitiedostot (kuva 21) sekä globaalit ja lokaalit suojamuuttujat (kuva 22). Tiedostot generoitiin, jonka jälkeen testiprojektin kansiosta löytyivät alkuperäisten lisäksi seuraavat tiedostot:

- test_file0_safe.c
- test_file1_safe.c
- safeguard.h
- safeguard.c.

Suojamuuttujien otsikkotiedostoon generoitui kaikkien aktiivisten suojamuuttujien ja tiedostojen define-direktiivit, tarkistusfunktioiden ja globaalien suojamuuttujien struktuurien esittelyt sekä otsikkotiedostolle määritelty lisäsisältö (liite 4). Suojamuuttujien tarkistus-

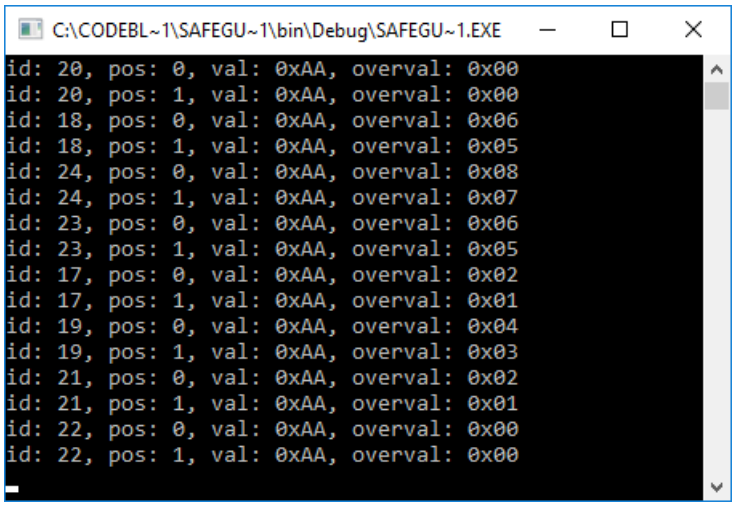
tiedostoon generoitui lokaalien ja globaalien suojamuuttujien tarkistusfunktiot, jotka kutsuvat kehittäjän määrittämää tarkistusfunktiota (liite 5). Testiprojektin alkuperäiset testitiedostot pudotettiin pois käännöksestä ja siihen lisättiin suojamuuttujia sisältävät testitiedostojen kopiot (kuva 39).



KUVA 39. Testiprojektin käännettävät tiedostot

6.3 Tulokset

Testiprojekti käännettiin ja ohjelma suoritettiin (kuva 40). Jokainen aiheutettu puskurin ylivuoto havaittiin.



KUVA 40. Testiprojektin suoritus

Liitteen 3 testitiedostossa staattiseksi määritetty lokaali puskuri on säilynyt staattisena generoituneissa tiedostossa (kuva 41). Struktuurin puskurille määritetyt arvot vastaavat alkuperäisen lähdekoodin puskurin arvoja.

```
static struct
{
#if defined(FILE_2) && defined(SAFEGUARD_24_POSITION_0)
volatile unsigned char safeguard_24_0;
#endif
unsigned int safeguard_buffer[5];
#if defined(FILE_2) && defined(SAFEGUARD_24_POSITION_1)
volatile unsigned char safeguard_24_1;
#endif
} safeguard_24 =
{
#if defined(FILE_2) && defined(SAFEGUARD_24_POSITION_0)
.safeguard_24_0=170,
#endif
.safeguard_buffer={1,2,TEST_VALUE,TEST_VALUE,3},
#if defined(FILE_2) && defined(SAFEGUARD_24_POSITION_1)
.safeguard_24_1=170,
#endif
};
```

KUVA 41. Staattisen lokaalin suojamuuttujan struktuurin esittely ja määritys

Saman testitiedoston staattiseksi määritetyn globaalin puskurin static-avainsana on poistettu (kuva 42).

```
struct safeguard_21_s safeguard_21=
{
#if defined(FILE_2) && defined(SAFEGUARD_21_POSITION_0)
.safeguard_21_0=170,
#endif
#if defined(FILE_2) && defined(SAFEGUARD_21_POSITION_1)
.safeguard_21_1=170,
#endif
};
```

KUVA 42. Globaalin suojamuuttujan struktuurin määritys

Kuvan 42 globaalin suojamuuttujan struktuurin esittely sijaitsee suojamuuttujien otsikkotiedostossa (kuva 43). Struktuurin puskurin koko vastaa alkuperäisen lähdekoodin puskurin kokoa.

```
#define SAFEGUARD_21_POSITION_0
#define SAFEGUARD_21_POSITION_1
struct safeguard_21_s
{
#if defined(FILE_2) && defined(SAFEGUARD_21_POSITION_0)
volatile unsigned char safeguard_21_0;
#endif
long safeguard_buffer[TEST_SIZE];
#if defined(FILE_2) && defined(SAFEGUARD_21_POSITION_1)
volatile unsigned char safeguard_21_1;
#endif
};
extern struct safeguard_21_s safeguard_21;
```

KUVA 43. Globaalin suojamuuttujan struktuurin esittely

IEEE 754 -standardi määrittää liukulukujen esityksen muistissa. 32-bittiselle liukuluvulle esitys jaetaan kolmeen osaan (taulukko 1). Ensimmäiselle osalle varataan yksi bitti, joka

ilmaisee liukuluvun etumerkin. Liukuluvun sisältämää osaa kutsutaan signifikantiksi, jolle varataan 23-bittiä. Loput kahdeksan bittiä varataan liukuluvun eksponentille. Eksponentin minimiarvo on −127 ja maksimiarvo on 128. Koska eksponentilla ei ole etumerkkiä ilmaisevaa bittiä, vastaa arvo 0 arvoa −127 ja arvo 255 vastaa arvoa 128. (12, s. 15.)

TAULUKKO 1. 32-bittisen liukuluvun esitys IEEE 754 -standardilla (12, s. 16)

b0	b1, b2, b3 … b8	b9, b10, b11 … b31
Etumerkki	Eksponentti	Signifikantti

Kaava 1 kuvaa liukulukua IEEE 754 -standardin mukaisesti, jossa *s* = etumerkki (0 tai 1), *f* = signifikantti ja *exp* = eksponentti (12, s. 16).

KAAVA 1

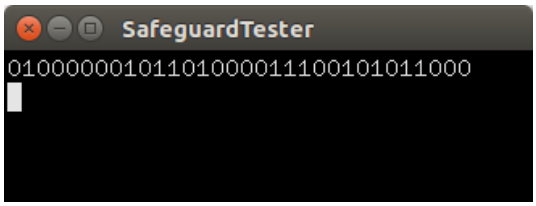
$$(-1)^s * (1.f)_2 * 2^{exp-127}$$

Liite 6 havainnollistaa kaavan 1 soveltamista ja esittää liukuluvun taulukon 1 mukaisesti. Kuvan 44 ohjelmalla tulostetaan float-muuttujan bitit eniten merkitsevästä bitistä alkaen. Float-muuttuja on määritetty liitteen 6 liukuluvun arvoon.

```
int main()
{
    float a = 5.632f;
    int *p = (int *)&a;
    int i;

    for (i = 31; i >= 0; i--)
    {
        printf("%d", (*p >> i) & 1);
    }

    printf("\n");
    return 0;
}
```

KUVA 44. Float-muuttujan sisällön tulostaminen

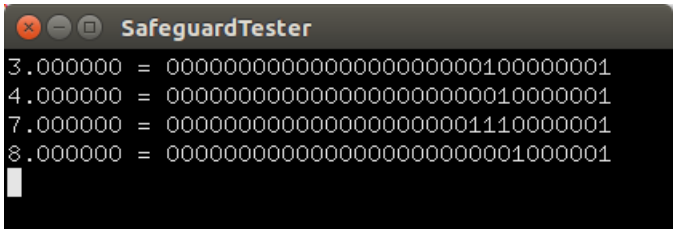
Kuvan 45 bitit täsmäävät liitteessä 6 lasketun IEEE 754 -standardin määrittämän liukulukuesityksen kanssa.



KUVA 45. Float-muuttujan sisältö muistissa MSB:stä alkaen

Muuttujan sisältöä voidaan varastoida muistiin kahdella tapaa. Big endian -muodossa eniten merkitsevä tavu sijoittuu varatun muistin alimpaan osoitteeseen, kun taas little endian -muodossa vähiten merkitsevä tavu sijoittuu varatun muistin alimpaan osoitteeseen. (13.)

Kuvan 44 ohjelmaa muutettiin tulostamaan testauksessa ylivuotaneet float-tyyppiset arvot. Float-arvojen bitit tulostetaan vähiten merkitsevästä bitistä alkaen. Kuvasta 46 voidaan päätellä miksi float-tyyppisiä puskureita suojanneet suojamuuttujat olivat arvoiltaan nollia ylivuotojen tapahduttua. Kaapatut ylivuotaneet tavut testauksessa ovat olleet signifikanttiosien kahdeksan viimeistä bittiä, jotka kuvassa 46 ovat nollia. Testaus suoritettiin Windows- ja Ubuntu-käyttöjärjestelmillä, jotka varastoivat muuttujat muistiin little endian -muodossa.



KUVA 46. Float-muuttujien sisältö muistissa LSB:stä alkaen

Ongelmana nolla-arvoisille suojamuuttujille ovat tyypillisten 16- ja 32-bittisten puskureiden ylivuodot, joissa ylivuotavan arvon ensimmäinen tavu on nolla. Testitulosten perusteella suojamuuttujaa ei tule määrittää arvoon nolla.

7 YHTEENVETO

Opinnäytetyön tarkoituksena oli kehittää menetelmä suojamuuttujien lisäämiseen lähdekoodiin puskureiden ylivuotojen havaitsemiseksi ajonaikana. Suojamuuttujien hallitsemiseen kehitettiin graafiseen käyttöliittymään perustuva sovellus, jolla hallitaan paikallista tietokantaa ja käsitellään tiedostoja. Sovellukselle jäi useita jatkokehityskohteita.

En käyttänyt käyttöliittymän suunnitteluun paljon aikaa, joten sen osalta työssä on paljon parannettavaa.

Sovelluksen käytön jatkokehityskohteena on suojamuuttujien lisäämisen nopeuttaminen. Usean suojamuuttujan lisääminen lähdekoodin saman istunnon aikana on hidasta. Ratkaisu tähän ongelmaan voisi olla yksinkertainen tekstitiedosto, johon kehittäjä lisää riveille suojamuuttujille annettavat parametrit. Sovellus käsittelisi tiedoston ja lisäisi suojamuuttujat tietokantaan. Toisaalta kun suojamuuttuja on kerran lisätty tietokantaan, voidaan se generoida aina uudestaan.

Itse suojamuuttujille näen kaksi jatkokehityskohdetta. Suojamuuttujan tietotyyppi tulisi voida määrittää sovelluksella. Lisäämällä suojamuuttujalle varattavan muistin määrää kasvatetaan ylivuodon havaitsemisen todennäköisyyttä. Suojamuuttujan arvo tulisi satunnaistaa, kuten Stack Smashing Protector -mallissa. Puskurin ylivuotava tavu prosessissa voi hyvinkin olla jokaisella ajokerralla sama. Jos suojamuuttujan arvo on jokaisessa generoinnissa sama ja puskurin ylivuotava tavu sattuu olemaan suojamuuttujan arvo, jää ylivuoto aina havaitsematta.

Vaikeinta työssä oli jatkuva sovelluksen lähdekoodin refaktorointi. Opin koko ajan tehokkaampia menetelmiä merkkijonojen käsittelyyn säännöllisillä lausekkeilla sekä järkevämpiä ratkaisuja käyttöliittymälle wxWidgets-kirjastolla. Opinnäytetyön aihe oli hyvin kiinnostava ja opin paljon uutta.

LÄHTEET

1. Seacord, R. 2006. Secure Coding in C and C++. Upper Saddle River: Addison-Wesley.

2. Holub, A. 1990. Compiler design in C. Saatavissa: http://holub.com/goodies/compiler/compilerDesignInC.pdf. Hakupäivä 18.1.2018.

3. Jones, R & Kelly, P. 1997. Backwards-compatible bounds checking for arrays and pointers in C programs. Proceedings of the 3rd International Workshop on Automatic Debugging; 1997 (AADEBUG-97). S. 13–26. Saatavissa: https://www.doc.ic.ac.uk/~phjk/Publications/BoundsCheckingForC.pdf. Hakupäivä 18.1.2018.

4. Hipp, R. 2016. About SQLite. SQLite. Saatavissa: https://www.sqlite.org/about.html. Hakupäivä 18.1.2018.

5. Hipp, R. 2016. Distinctive Features Of SQLite. SQLite. Saatavissa: https://www.sqlite.org/different.html. Hakupäivä 18.1.2018.

6. Hipp, R. 2016. SQLite is a Self Contained System. SQLite. Saatavissa: https://www.sqlite.org/selfcontained.html. Hakupäivä 18.1.2018.

7. Smart, J., Zeitlin, V., Dunn, R., Csomor, S., Petty, B., Montorsi, F. & Roebling, R. 2016. Introduction. wxWidgets. Saatavissa: http://docs.wxwidgets.org/trunk/page_introduction.html. Hakupäivä 18.1.2018.

8. Smart, J., Zeitlin, V., Dunn, R., Csomor, S., Petty, B., Montorsi, F. & Roebling, R. 2016. wxWindows Library License. wxWidgets. Saatavissa: http://docs.wxwidgets.org/trunk/page_copyright_wxlicense.html. Hakupäivä 18.1.2018.

9. Telle, U. 2017. WxSQLite3 – a lightweight wrapper for SQLite. Saatavissa: https://github.com/utelle/wxsqlite3. Hakupäivä 18.1.2018.

10. Telle, U. 2017. LICENSE. Saatavissa: https://github.com/utelle/wxsqlite3/blob/master/LICENSE. Hakupäivä 18.1.2018.

11. Smart, J., Zeitlin, V., Dunn, R., Csomor, S., Petty, B., Montorsi, F. & Roebling, R. 2016. Regular Expressions. wxWidgets. Saatavissa: http://docs.wxwidgets.org/trunk/overview_resyntax.html. Hakupäivä 18.1.2018.

12. Rajaraman, V. 2016. IEEE Standard for Floating Point Numbers. Resonance – Journal of Science Education vol. 21, nro 1. S. 11–30. Saatavissa: http://www.ias.ac.in/article/fulltext/reso/021/01/0011-0030. Hakupäivä 18.1.2018.

13. Adiga, H. 2007. Writing endian-independent code in C. IBM developerWorks. Saatavissa: https://www.ibm.com/developerworks/aix/library/au-endianc/index.html. Hakupäivä 18.1.2018.

LIITTEET

Liite 1 Tietokannan luontilauseet

Liite 2 Ensimmäisen testitiedoston lähdekoodi

Liite 3 Toisen testitiedoston lähdekoodi

Liite 4 Suojamuuttujien otsikkotiedoston lähdekoodi

Liite 5 Suojamuuttujien tarkistustiedoston lähdekoodi

Liite 6 Esimerkki IEEE 754 -standardista

```
CREATE TABLE project
(
        id INTEGER PRIMARY KEY,
        headerpath TEXT NOT NULL,
        checkerpath TEXT NOT NULL,
        includes TEXT,
        name TEXT NOT NULL unique
);

CREATE TABLE file
(
        id INTEGER PRIMARY KEY,
        name TEXT NOT NULL,
        path TEXT NOT NULL,
        enabled INTEGER NOT NULL,
        projectid INTEGER NOT NULL,
        FOREIGN KEY(projectid) REFERENCES project(id) ON DELETE CASCADE
);

CREATE TABLE safeguard
(
        id INTEGER PRIMARY KEY AUTOINCREMENT
);

CREATE TABLE globalsafeguard
(
        id INTEGER PRIMARY KEY,
        datatype TEXT NOT NULL,
        variable TEXT NOT NULL,
        value INTEGER NOT NULL,
        above INTEGER NOT NULL,
        below INTEGER NOT NULL,
        status INTEGER NOT NULL,
        fileid INTEGER NOT NULL,
        safeguardid INTEGER NOT NULL,
        FOREIGN KEY(fileid) REFERENCES file(id) ON DELETE CASCADE,
        FOREIGN KEY(safeguardid) REFERENCES safeguard(id) ON DELETE CASCADE
);

CREATE TABLE localsafeguard
(
        id INTEGER PRIMARY KEY,
        datatype TEXT NOT NULL,
        variable TEXT NOT NULL,
        value INTEGER NOT NULL,
        function TEXT NOT NULL,
        above INTEGER NOT NULL,
        below INTEGER NOT NULL,
        status INTEGER NOT NULL,
        fileid INTEGER NOT NULL,
        safeguardid INTEGER NOT NULL,
        FOREIGN KEY(fileid) REFERENCES file(id) ON DELETE CASCADE,
        FOREIGN KEY(safeguardid) REFERENCES safeguard(id) ON DELETE CASCADE
);
```

```c
#include "test_file.h"

//<safeguard_global_space>
int buffer_0_0[5];
static unsigned int buffer_0_1[] = {TEST_VALUE, 1, 2, TEST_VALUE};
//</safeguard_global_space>

//<safeguard_local_space='test_file0_func'>
void test_file0_func(void)
{
    static long buffer_0_2[TEST_SIZE];
    float buffer_0_3[] = {0.0, 0.1, 0.2, 0.3};

    buffer_0_2[1] = 12;
    if (buffer_0_2[1] > 10)
    {
        buffer_0_0[3] = 53;
        buffer_0_3[0] = (float) buffer_0_1[2] / buffer_0_0[3];
    }

#if defined(TEST_OVERFLOW)
    buffer_0_0[sizeof(buffer_0_0) / sizeof(int)] = 1;
    buffer_0_0[-1] = 2;
    buffer_0_1[sizeof(buffer_0_1) / sizeof(unsigned int)] = 3;
    buffer_0_1[-1] = 4;
    buffer_0_2[TEST_SIZE] = 5;
    buffer_0_2[-1] = 6;
    buffer_0_3[sizeof(buffer_0_3) / sizeof(float)] = 7;
    buffer_0_3[-1] = 8;
#endif

    //<safeguard_checkpoint>
}
//</safeguard_local_space='test_file0_func'>
```

```c
#include "test_file.h"

//<safeguard_global_space>
static long buffer_1_0[TEST_SIZE];
float buffer_1_1[] = {1.0, 1.1, 1.2, 1.3};
//</safeguard_global_space>

//<safeguard_local_space='test_file1_func'>
void test_file1_func(void)
{
    int buffer_1_2[12];
    static unsigned int buffer_1_3[] = {1, 2, TEST_VALUE, TEST_VALUE, 3};

    buffer_1_0[2] = 53;
    if (buffer_1_0[2] > 20)
    {
        buffer_1_2[7] = 25;
        buffer_1_1[2] = (float) buffer_1_3[3] / buffer_1_2[7];
    }

#if defined(TEST_OVERFLOW)
    buffer_1_0[TEST_SIZE] = 1;
    buffer_1_0[-1] = 2;
    buffer_1_1[sizeof(buffer_1_1) / sizeof(float)] = 3;
    buffer_1_1[-1] = 4;
    buffer_1_2[sizeof(buffer_1_2) / sizeof(int)] = 5;
    buffer_1_2[-1] = 6;
    buffer_1_3[sizeof(buffer_1_3) / sizeof(unsigned int)] = 7;
    buffer_1_3[-1] = 8;
#endif

    //<safeguard_checkpoint>
}
//</safeguard_local_space='test_file1_func'>
```

```
#include "test_file.h"
void safeguard_user_function(unsigned char overval, int pos, int id, unsigned char val);
void global_safeguard_check(void);
void local_safeguard_check(unsigned char overval, int pos, int id, unsigned char val);
#define FILE_1
#define SAFEGUARD_18_POSITION_0
#define SAFEGUARD_18_POSITION_1
#define SAFEGUARD_20_POSITION_0
#define SAFEGUARD_20_POSITION_1
#define SAFEGUARD_17_POSITION_0
#define SAFEGUARD_17_POSITION_1
struct safeguard_17_s
{
#if defined(FILE_1) && defined(SAFEGUARD_17_POSITION_0)
volatile unsigned char safeguard_17_0;
#endif
int safeguard_buffer[5];
#if defined(FILE_1) && defined(SAFEGUARD_17_POSITION_1)
volatile unsigned char safeguard_17_1;
#endif
};
extern struct safeguard_17_s safeguard_17;
#define SAFEGUARD_19_POSITION_0
#define SAFEGUARD_19_POSITION_1
struct safeguard_19_s
{
#if defined(FILE_1) && defined(SAFEGUARD_19_POSITION_0)
volatile unsigned char safeguard_19_0;
#endif
unsigned int safeguard_buffer[4];
#if defined(FILE_1) && defined(SAFEGUARD_19_POSITION_1)
volatile unsigned char safeguard_19_1;
#endif
};
extern struct safeguard_19_s safeguard_19;
#define FILE_2
#define SAFEGUARD_23_POSITION_0
#define SAFEGUARD_23_POSITION_1
#define SAFEGUARD_24_POSITION_0
#define SAFEGUARD_24_POSITION_1
#define SAFEGUARD_21_POSITION_0
#define SAFEGUARD_21_POSITION_1
struct safeguard_21_s
{
#if defined(FILE_2) && defined(SAFEGUARD_21_POSITION_0)
volatile unsigned char safeguard_21_0;
#endif
long safeguard_buffer[TEST_SIZE];
#if defined(FILE_2) && defined(SAFEGUARD_21_POSITION_1)
volatile unsigned char safeguard_21_1;
#endif
};
extern struct safeguard_21_s safeguard_21;
#define SAFEGUARD_22_POSITION_0
#define SAFEGUARD_22_POSITION_1
struct safeguard_22_s
{
#if defined(FILE_2) && defined(SAFEGUARD_22_POSITION_0)
volatile unsigned char safeguard_22_0;
#endif
float safeguard_buffer[4];
#if defined(FILE_2) && defined(SAFEGUARD_22_POSITION_1)
volatile unsigned char safeguard_22_1;
#endif
};
extern struct safeguard_22_s safeguard_22;
```

```
#include "safeguard.h"
void local_safeguard_check(unsigned char overval, int pos, int id, unsigned char val)
{
if (val != overval)
safeguard_user_function(overval, pos, id, val);
}
void global_safeguard_check(void)
{
#if defined(FILE_1) && defined(SAFEGUARD_17_POSITION_0)
if (safeguard_17.safeguard_17_0 != 170)
safeguard_user_function(safeguard_17.safeguard_17_0, 0, 17, 170);
#endif
#if defined(FILE_1) && defined(SAFEGUARD_17_POSITION_1)
if (safeguard_17.safeguard_17_1 != 170)
safeguard_user_function(safeguard_17.safeguard_17_1, 1, 17, 170);
#endif
#if defined(FILE_1) && defined(SAFEGUARD_19_POSITION_0)
if (safeguard_19.safeguard_19_0 != 170)
safeguard_user_function(safeguard_19.safeguard_19_0, 0, 19, 170);
#endif
#if defined(FILE_1) && defined(SAFEGUARD_19_POSITION_1)
if (safeguard_19.safeguard_19_1 != 170)
safeguard_user_function(safeguard_19.safeguard_19_1, 1, 19, 170);
#endif
#if defined(FILE_2) && defined(SAFEGUARD_21_POSITION_0)
if (safeguard_21.safeguard_21_0 != 170)
safeguard_user_function(safeguard_21.safeguard_21_0, 0, 21, 170);
#endif
#if defined(FILE_2) && defined(SAFEGUARD_21_POSITION_1)
if (safeguard_21.safeguard_21_1 != 170)
safeguard_user_function(safeguard_21.safeguard_21_1, 1, 21, 170);
#endif
#if defined(FILE_2) && defined(SAFEGUARD_22_POSITION_0)
if (safeguard_22.safeguard_22_0 != 170)
safeguard_user_function(safeguard_22.safeguard_22_0, 0, 22, 170);
#endif
#if defined(FILE_2) && defined(SAFEGUARD_22_POSITION_1)
if (safeguard_22.safeguard_22_1 != 170)
safeguard_user_function(safeguard_22.safeguard_22_1, 1, 22, 170);
#endif
}
```

```
Esimerkki) Liukuluku 5.632 esitettynä IEEE 754 -standardin
mukaisesti 32-bittisenä

5 = 101

0.632 * 2 = 1.264 // bit = 1
0.264 * 2 = 0.528 // bit = 0
0.528 * 2 = 1.056 // bit = 1
0.056 * 2 = 0.112 // bit = 0
0.112 * 2 = 0.224 // bit = 0
0.224 * 2 = 0.448 // bit = 0
0.448 * 2 = 0.896 // bit = 0
0.896 * 2 = 1.792 // bit = 1
0.792 * 2 = 1.584 // bit = 1
0.584 * 2 = 1.168 // bit = 1
0.168 * 2 = 0.336 // bit = 0
0.336 * 2 = 0.672 // bit = 0
0.672 * 2 = 1.344 // bit = 1
0.344 * 2 = 0.688 // bit = 0
0.688 * 2 = 1.377 // bit = 1
0.377 * 2 = 0.754 // bit = 0
0.754 * 2 = 1.508 // bit = 1
0.508 * 2 = 1.016 // bit = 1
0.016 * 2 = 0.031 // bit = 0
0.031 * 2 = 0.062 // bit = 0
0.062 * 2 = 0.125 // bit = 0
0.125 * 2 = 0.250 // bit = 0
0.250 * 2 = 0.500 // bit = 0

101.10100001110010101100000
= 10.11010000111001010110000 * 2
= 1.01101000011100101011000 * 2^2

f = 01101000011100101011000 (23 bittiä)

exp - 127 = 2
exp = 129
exp = 10000001 (8 bittiä)

5.632 > 0, joten s = 0

0 10000001 01101000011100101011000
| |______| |_______________________|
| |        |
| |        signifikantti (f)
| eksponentti (exp)
etumerkki (s)
```